# Intel® Math Kernel Library for the Linux* OS

User's Guide

***August 2008***

Document Number: 314774-007US

World Wide Web: http://developer.intel.com

| Version | Version Information | Date |
|---|---|---|
| -001 | Original issue. Documents Intel® Math Kernel Library (Intel® MKL) 9.0 gold release. | September 2006 |
| -002 | Documents Intel® MKL 9.1 beta release. “Getting Started”, “LINPACK and MP LINPACK Benchmarks” chapters and “Support for Third-Party and Removed Interfaces” appendix added. Existing chapters extended. Document restructured. List of examples added. | January 2007 |
| -003 | Documents Intel® MKL 9.1 gold release. Existing chapters extended. Document restructured. More aspects of ILP64 interface discussed. Section “Configuring the Eclipse* IDE CDT to Link with Intel MKL” added to chapter 3. Cluster content is organized into one separate chapter 9 “Working with Intel® Math Kernel Library Cluster Software” and restructured, appropriate links added. | June 2007 |
| -004 | Documents Intel® MKL 10.0 Beta release. Layered design model has been described in chapter 3 and the content of the entire book adjusted to the model. Automation of setting environment variables at startup has been described in chapter 4. New Intel MKL threading controls have been described in chapter 6. The User’s Guide for Intel MKL merged with the one for Intel MKL Cluster Edition to reflect consolidation of the respective products. | September 2007 |
| -005 | Documents Intel® MKL 10.0 Gold release. Configuring of Eclipse CDT 4.0 to link with Intel MKL has been described in chapter 3. Intel® Compatibility OpenMP* run-time compiler library (`libiomp`) has been described. | October 2007 |
| -006 | Documents Intel® MKL 10.1 beta release. Information on dummy libraries in Table "High-level directory structure" has been further detailed. Information on the Intel MKL configuration file removed. Section “Accessing Man Pages” has been added to chapter 3. Section "Support for Boost uBLAS Matrix-Matrix Multiplication" has been added to chapter 7. Chapter “Getting Assistance for Programming in the Eclipse* IDE” has been added. | May 2008 |
| -007 | Documents Intel® MKL 10.1 gold release. Linking examples for IA-32 architecture and section "Linking with Computational Libraries" have been added to chapter 5. Integration of DSS/PARDISO into the layered structure has been-documented. Two Fortran code examples have been added. | August 2008 |

# Contents

## List of Tables

## List of Examples

## List of Figures

# Overview 1

Intel® Math Kernel Library (Intel® MKL) offers highly optimized, thread-safe math routines for science, engineering, and financial applications that require maximum performance.

## Technical Support

Intel provides a support web site, which contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel® MKL support website at http://www.intel.com/software/products/support/ .

## About This Document

*Intel MKL Reference Manual* provides *reference* information on routine functionalities, parameter descriptions, interfaces, calling syntaxes, and return values. The *Intel MKL User's Guide* contains *usage information*, which goes into more technical detail than reference information. Usage information explains more specifics of routine calls, especially, in mixed-language programming, and additionally covers the organization, configuration, performance, and accuracy of Intel MKL.

This guide focuses on the usage information needed to call Intel MKL routines from user's applications running on the Linux* OS. The document describes features particular to Linux usage of Intel MKL, along with OS-independent features.

This guide contains usage information for all Intel MKL function domains, listed in Table A-1 (in Appendix A).

This document assumes you have completed the installation of Intel MKL on your system. If you have not completed the installation, see *Intel® Math Kernel Library Installation Guide* (file `Install.txt`).

This guide should be used in conjunction with the latest version of the *Intel® Math Kernel Library for Linux* Release Notes* document to reference how to use the library in your application.

## Purpose

Intel® Math Kernel Library for Linux* User's Guide is intended to assist you in mastering the usage of the Intel MKL on Linux. In particular, it

- Describes post-installation steps to help you start using the library
- Shows you how to configure the library with your development environment
- Acquaints you with the library structure
- Explains how to link your application to the library and provides simple usage scenarios
- Describes how to code, compile, and run your application with Intel MKL for Linux.

## Audience

The guide is intended for Linux programmers with beginner to advanced experience in software development.

## Document Organization

The document contains the following chapters and appendices:

| | |
|---|---|
| Chapter 1 | Overview. Introduces the concept of the Intel MKL usage information; describes the document's purpose and organization as well as explains notational conventions. |
| Chapter 2 | Getting Started. Describes post-installation steps and gives information needed to start using Intel MKL after its installation. |
| Chapter 3 | Intel® Math Kernel Library Structure. Discusses the structure of the Intel MKL directory after installation at different levels of detail as well as the library versions and parts. |
| Chapter 4 | Configuring Your Development Environment. Explains how to configure Intel MKL with your development environment. |
| Chapter 5 | Linking Your Application with Intel® Math Kernel Library. Compares static and dynamic linking models; describes the general link line syntax to be used for linking with Intel MKL libraries; |

|  |  |
|---|---|
|  | explains which libraries should be linked with your application for your particular platform and function domain; discusses how to build custom dynamic libraries. |
| Chapter 6 | Managing Performance and Memory. Discusses Intel MKL threading; shows coding techniques and gives hardware configuration tips for improving performance of the library; explains features of the Intel MKL memory management and, in particular, shows how to replace memory functions that the library uses by default with your own ones. |
| Chapter 7 | Language-specific Usage Options. Discusses mixed-language programming and the use of language-specific interfaces. |
| Chapter 8 | Coding Tips. Presents coding tips that may be helpful to your specific needs. |
| Chapter 9 | Working with Intel® Math Kernel Library Cluster Software. Discusses usage of ScaLAPACK and Cluster FFTs; in particular, describes linking of your application with these function domains, including C- and Fortran-specific linking examples; gives information on the supported MPI. |
| Chapter 10 | Getting Assistance for Programming in the Eclipse* IDE. Discusses Intel MKL features that software engineers can benefit from when working in the Eclipse* IDE. |
| Chapter 11 | LINPACK and MP LINPACK Benchmarks. Describes the Intel® Optimized LINPACK Benchmark for Linux* and Intel® Optimized MP LINPACK Benchmark for Clusters. |
| Appendix A | Intel® Math Kernel Library Language Interfaces Support. Summarizes information on language interfaces that Intel MKL provides for each function domain, including the respective header files. |
| Appendix B | Support for Third-Party Interfaces. Describes in brief some interfaces that Intel MKL supports. |

The document also includes an Index.

## Term and Notational Conventions

The following term is used in the manual in reference to the operating system:

|  |  |
|---|---|
| Linux* OS | This term refers to information that is valid on all supported Linux* operating systems. |

The following notation is used in reference to Intel MKL directories:

| | |
|---|---|
| `<mkl_directory>` | The main directory to which Intel MKL is installed. Should be substituted with the specific pathname in the configuring, linikng, and building instructions: |

**Table 1-1 Notational conventions**

| | |
|---|---|
| *Italic* | Italic is used for emphasis and also indicates document names in body text, for example: see *Intel MKL Reference Manual* |
| `Monospace lowercase` | Indicates filenames, directory names and pathnames, for example: `libmkl_core.a` , `/opt/intel/mkl/10.1.0.004` |
| `Monospace lowercase mixed with uppercase` | Indicates commands and command-line options, for example: `icc myprog.c -L$MKLPATH -I$MKLINCLUDE -lmkl -lguide -lpthread ;` C/C++ code fragments, for example: `a = new double [SIZE*SIZE];` |
| `UPPERCASE MONOSPACE` | Indicates system variables, for example, `$MKLPATH` |
| *`Monospace italic`* | Indicates a parameter in discussions: routine parameters, for example, *`lda`*; makefile parameters, for example, *`functions_list`*; etc. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, *`<mkl directory>`*. Substitute one of these items for the placeholder. |
| [ items ] | Square brackets indicate that the items enclosed in brackets are optional. |
| { item \| item } | Braces indicate that only one of the items listed between braces should be selected. A vertical bar ( \| ) separates the items |

# Getting Started 2

This chapter helps you start using the Intel® Math Kernel Library (Intel® MKL) for the Linux* OS by giving you some basic information and describing post-installation steps.

## Checking Your Installation

After installing Intel MKL, verify that the library has been properly installed and configured.

1. Check that the directory you chose for the installation has been created. The Intel MKL *default* installation directory may be one of the following:
   - `/opt/intel/mkl/RR.r.y.xxx`, where `RR.r` is the version number, `y` is the release-update number and `xxx` is the package number, for example, `/opt/intel/mkl/10.1.0.004`
   - *`<Intel Compiler Pro directory>`*`/mkl`, where *`<Intel Compiler Pro directory>`* is the installation directory for Intel® C++ Compiler Professional Edition or Intel® Fortran Compiler Professional Edition, for example, `/opt/intel/Compiler/11.0.015/mkl`.
2. If you choose to keep multiple versions of Intel MKL installed on your system, update build scripts so that they point to the *desired* version.
3. Check that the following six files are placed in the `tools/environment` directory:

   ```
   mklvars32.sh
   mklvars32.csh
   mklvarsem64t.sh
   mklvarsem64t.csh
   mklvars64.sh
   mklvars64.csh
   ```

   You can use these files to set environmental variables, such as `INCLUDE`, `LD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, and `FPATH`, in the current user shell.

## Obtaining Version Information

Intel MKL provides Intel MKL provides two methods for obtaining information about the current library (for example, the version number). The function `MKLGetVersionString` extracts a version string, and the function `MKLGetVersion` can be used to obtain the `MKLVersion` structure which also contains the version information. For the function descriptions and calling syntax, see the *Support Functions* chapter in the *Intel MKL Reference Manual*. Sample programs for extracting version information are provided in the `examples/versionquery` directory. A makefile is also provided to automatically build the examples and output summary files containing the version information.

## Compiler Support

Intel supports Intel® MKL for use only with compilers identified in the *Release Notes*. However, the library has been successfully used with other compilers as well.

When using the CBLAS interface, the header file `mkl.h` will simplify program development, since it specifies enumerated values as well as prototypes for all the functions. The header determines if the program is being compiled with a C++ compiler and, if so, the included file will be correct for use with C++ compilation.

Starting with Intel MKL 9.1, full support is provided for the GNU gfortran* compiler, which differs from the Intel® Fortran Compiler in calling conventions for functions that return complex data. Absoft* Fortran compilers are supported as well. For usage specifics of the Absoft compilers, see Linking with the Absoft* compilers in chapter 5.

## Using Intel MKL Code Examples

Intel MKL package includes code examples, located in the `examples` subdirectory of the Intel MKL installation directory. The examples provide the most direct and immediate way for you to find out:

- Whether Intel MKL is working on your system
- How you should call the library
- How to link the library.

The examples are grouped in subdirectories mainly by Intel MKL function domains and programming languages. For instance, subdirectory `examples/spblas` contains Sparse BLAS examples, and subdirectory `examples/vmlc` contains VML examples in C. Source code of the examples is in the next level `sources` subdirectory.

To compile, build, and run the examples, use the makefile provided. For information on how to use it, refer to the makefile header.

See also:

High-level Directory Structure in chapter 3.

## Before You Begin Using Intel MKL

Before you get started using the Intel MKL, sorting out a few important basic concepts will greatly help you get off to a good start.

The table below summarizes some important things to think of before you start using Intel MKL.

**Table 2-1 What you need to know before you get started**

| | |
|---|---|
| Target platform | Identify the architecture of your target machine:<br>• IA-32<br>• Intel® 64<br>• IA-64 (Itanium® processor family)<br>**Reason.** Intel MKL libraries, which you should link with your application, are located in directories corresponding to your particular architecture (see Intel® MKL Versions). So, you should provide proper paths in your link lines (see Linking Examples). To configure your development environment for the use with Intel MKL, set your environment variables using the script corresponding to your architecture (see Setting Environment Variables). |
| Mathematical problem | Identify all Intel MKL function domains that problems you are solving require:<br>• BLAS<br>• Sparse BLAS<br>• LAPACK<br>• PBLAS<br>• ScaLAPACK<br>• Sparse Solver routines<br>• Vector Mathematical Library functions<br>• Vector Statistical Library functions<br>• Fourier Transform functions (FFT)<br>• Cluster FFT<br>• PBLAS<br>• Trigonometric Transform routines<br>• Poisson, Laplace, and Helmholtz Solver routines<br>• Optimization (Trust-Region) Solver routines<br>• GMP* arithmetic functions<br>**Reason.** The function domain you intend to use narrows the search in the *Reference Manual* for specific routines you need. Additionally, the link line that you use to link your application with Intel MKL cluster software depends on the function domains you intend to employ (see Working with Intel® Math Kernel Library Cluster Software). Coding tips may also depend on the function domain (see Tips and Techniques to Improve Performance). |

**Table 2-1 What you need to know before you get started** (continued)

| | |
|---|---|
| Programming language | Though Intel MKL provides support for both Fortran and C/C++ programming, not all the function domains support a particular language environment, for example, C/C++ or Fortran90/95. Identify the language interfaces that your function domains support (see Intel® Math Kernel Library Language Interfaces Support).<br>**Reason.** In case your function domain does not directly support the needed environment, you can use mixed-language programming (see Mixed-language programming with Intel® MKL).<br>For a list of language-specific interface libraries and modules and an example how to generate them, see also Using Language-Specific Interfaces with Intel® MKL. |
| Threading model | Select among the following options how you are going to thread your application:<br>• Your application is already threaded<br>• You may want to use the Intel® threading capability, that is, Compatibility OpenMP* run-time library (`libiomp`) or Legacy OpenMP* run-time library (`libguide`), or a threading capability provided by a third-party compiler<br>• You do not want to thread your application.<br>**Reason.** By default, the OpenMP* software sets the number of threads that Intel MKL uses. If you need a different number, you have to set it yourself using one of the available mechanisms. For more information, and especially, how to avoid conflicts in the threaded execution environment, see Using Intel® MKL Parallelism. Additionally, the compiler that you use to thread your application determines which threading library you should link with your application (see Linking Examples). |
| Linking model | Decide which linking model is appropriate for linking your application with Intel MKL libraries:<br>• Static<br>• Dynamic<br>**Reason.** For information on the benefits of each linking model, link command syntax and examples, link libraries, and other linking topics, like how to save disk space by creating a custom dynamic library, see Linking Your Application with Intel® Math Kernel Library. |
| MPI used | **Reason:** To link your application with ScaLAPACK and/or Cluster FFT, the libraries corresponding to your particular MPI should be included in the link line (see Working with Intel® Math Kernel Library Cluster Software). |

# *Intel® Math Kernel Library Structure* 3

The chapter discusses the structure of the Intel® Math Kernel Library (Intel® MKL), including the Intel MKL directory structure, as well as the library versions and parts.

Starting with version 10.1, Intel MKL employs a layered model to streamline the library structure, reduce its size, and add usage flexibility.

See also: Layered Model Concept.

## High-level Directory Structure

Table 3-1 shows a high-level directory structure of Intel MKL after installation.

**Table 3-1 High-level directory structure**

| Directory | Comment |
|---|---|
| `<mkl directory>` | Intel MKL main directory |
| `<mkl directory>/benchmarks/linpack` | Contains an OMP version of the LINPACK benchmark |
| `<mkl directory>/benchmarks/mp_linpack` | Contains an MPI version of the LINPACK benchmark |
| `<mkl directory>/doc` | Contains the Intel MKL documentation |
| `<mkl directory>/examples` | Contains source code, data, and makefiles for examples |
| `<mkl directory>/include` | Contains INCLUDE files for library routines as well as for test and example programs |
| `<mkl directory>/interfaces/blas95` | Contains Fortran 95 wrappers for BLAS and a makefile to build the library |
| `<mkl directory>/interfaces/fftw2x_cdft` | Contains wrappers for MPI FFTW version 2.x (for complex 1D transforms) to call Intel MKL Cluster FFT interface |

**Table 3-1 High-level directory structure** (continued)

| Directory | Comment |
|---|---|
| `<mkl directory>/interfaces/fftw2xc` | Contains wrappers for FFTW version 2.x (C interface) to call Intel MKL FFTs |
| `<mkl directory>/interfaces/fftw2xf` | Contains wrappers for FFTW version 2.x (Fortran interface) to call Intel MKL FFTs |
| `<mkl directory>/interfaces/fftw3xc` | Contains wrappers for FFTW version 3.x (C interface) to call Intel MKL FFTs |
| `<mkl directory>/interfaces/fftw3xf` | Contains wrappers for FFTW version 3.x (Fortran interface) to call Intel MKL FFTs |
| `<mkl directory>/interfaces/lapack95` | Contains Fortran 95 wrappers for LAPACK and a makefile to build the library |
| `<mkl directory>/lib/32` | Contains static libraries and shared objects for IA-32 architecture |
| `<mkl directory>/lib/64` | Contains static libraries and shared objects for IA-64 architecture (Itanium® processor family) |
| `<mkl directory>/lib/em64t` | Contains static libraries and shared objects for Intel® 64 architecture (formerly, Intel® EM64T) |
| `<mkl directory>/man/man3` | Contains man pages for Intel MKL functions |
| `<mkl directory>/tests` | Contains source and data files for tests |
| `<mkl directory>/tools/builder` | Contains tools for creating custom dynamically linkable libraries |
| `<mkl directory>/tools/environment` | Contains shell scripts to set environmental variables in the user shell |
| `<mkl directory>/tools/plugins/com.intel.mkl.help` | Contains an Eclipse* IDE plug-in with Intel MKL Reference Manual in WebHelp format. See `mkl_documentation.htm` for comments. |
| `<mkl directory>/tools/support` | Contains a utility for reporting the package ID and license key information to Intel® Premier Support |

## Layered Model Concept

The Intel Math Kernel Library has long had a structure that is not visible to the user except for the 32-bit version for Windows* OS. In that case, two interface libraries are provided, and you must select one of them at run time. Both libraries are relatively small and independent of a particular processor based on IA-32 architecture, and the major part of Intel MKL, which is independent of the interface, is not duplicated. Thus, by means of the interface libraries Intel MKL supports two different compiler interface standards without greatly increasing the library size.

Starting with release 10.0, Intel MKL is extending this approach to support a richer set of circumstances: compilers and threading.

**Interfaces.** On Linux systems based on IA-64 architecture, the Intel® Fortran Compiler returns complex values differently than gnu and some other compilers. Rather than duplicate the library for these differences, separate interface libraries are provided to support compiler differences while constraining the size of the library. Similarly, LP64 can be supported on top of ILP64 through an interface. Moreover, interface libraries are provided to support legacy supercomputers where single precision means 64-bit arithmetic.

**Threading.** For efficiency reasons, Intel MKL employs function-level threading throughout the library rather than loop-level threading. Consequently, all threading can be constrained to a relatively small set of functions and collected into a library. All references to compiler-specific run-time libraries are generated in these functions. By compiling them with different compilers and providing a threading library layer Intel MKL can work in programs threaded with Intel compilers and other supported threading compilers. A non-threaded library version can also be obtained by turning off threading when compiling the threading library layer because all threading is provided through OpenMP* technology.

**Computation.** For any given processor family (processors based on IA-32, IA-64, or Intel® 64 architecture), a single computational library is used for all interfaces and threading layers because there is no parallelism in the computational layer.

**Run-time library (RTL).** The last layer provides RTL support. Not all RTLs are delivered with Intel MKL. The only RTLs provided, except those that are relevant to the Intel MKL cluster software, are Intel® compiler based RTLs: Intel® Compatibility OpenMP* run-time compiler library (`libiomp`) and Intel® Legacy OpenMP* run-time compiler library (`libguide`). To thread using third-party threading compilers, you can employ Threading layer libraries or use the compatibility library in the appropriate circumstances.

## Layers

There are four essential parts of the library:

1. Interface layer
2. Threading layer
3. Computational layer
4. Compiler Support RTL layer (RTL layer, for brevity).

**Interface Layer.** This layer provides matching between compiled code of your application and the threading and/or computational parts of the library. This layer provides:

- An LP64 interface to Intel MKL ILP64 software
  (see Support for ILP64 Programming for details)
- A means of dealing with the way different compilers return function values

- A means of mapping between single-precision names and double-precision names in applications that employ ILP64, such as Cray-style naming.

**Threading Layer.** This layer provides a way for threaded Intel MKL to share supported compiler threading. The layer also provides for a sequential version of the library. What was internal to the library previously, now is essentially exposed in the threading layer. This layer is compiled for different environments (threaded or sequential) and compilers (Intel, gnu, and so on).

**Computational Layer.** This is the heart of Intel MKL and has only one variant for any processor/operating system family, such as 32-bit Intel® processors on a 32-bit operating system. The computational layer accommodates multiple architectures through identification of the architecture or architectural feature and chooses the appropriate binary code at execution. Intel MKL may be thought of as the large computational layer that is unaffected by different computational environments. Then, as it has no RTL requirements, RTLs refer not to the computational layer but to one of the layers above it: the Interface layer or Threading layer. The most likely case is matching the threading layer with the RTL layer.

**RTL Layer.** This layer has run-time library support functions. For example, `libiomp` and `libguide` are RTLs providing threading support for the OpenMP* threading in Intel MKL.

See also the "Linking Examples" section in chapter 5.

## Sequential Version of the Library

Starting with release 9.1, the Intel MKL package provides support for sequential (non-threaded) version of the library. It requires no RTL layer, that is, no Compatibility OpenMP* or Legacy OpenMP* run-time library, and does not respond to the environment variable `OMP_NUM_THREADS`. This version of Intel MKL runs unthreaded code. However, it is thread-safe[1], which means that you can use it in a parallel region from your own OpenMP* code. You should use sequential version only if you have a particular reason not to use Intel MKL threading. The sequential version (layer) may be helpful when using Intel MKL with programs threaded with non-Intel compilers or in other situations where you may, for various reasons, need a non-threaded version of the library. To obtain sequential version of Intel MKL, in the Threading layer, choose the `*sequential.*` library to link.

Note that the sequential library depends on the POSIX threads library (`pthread`), which is used to make the Intel MKL software thread-safe and should be included in the link line.

See also:

Directory Structure in Detail

Using Intel® MKL Parallelism in chapter 6

1. Except for LAPACK deprecated routines ?lacon, ?lasq3, and ?lasq4.

Avoiding Conflicts in the Execution Environment in chapter 6

Linking Examples in chapter 5.

## Support for ILP64 Programming

The terms "LP64" and "ILP64" are used for certain historical reasons and due to the programming models philosophy described here: http://www.unix.org/version2/whatsnew/lp64_wp.html .

Intel MKL ILP64 libraries do not completely follow the programming models philosophy. However, the general idea is the same: use 64-bit integer type for indexing huge arrays (arrays with more than $2^{31}$-1 elements).

It is up to you to choose which interface to use. You should definitely choose LP64 interface for compatibility with the previous Intel MKL versions because "LP64" is just a new name for the only interface that the Intel MKL versions lower than 9.1 provided. You should definitely choose the ILP64 interface if your application uses Intel MKL for calculations with huge data arrays (of more than $2^{31}$-1 elements) or the library may be used so in future.

The LP64 and ILP64 interfaces are supported in the Interface layer. Once the appropriate library in the Interface layer is selected (see Directory Structure in Detail), all libraries below the Interface layer are compiled using the chosen interface.

As the differences between the ILP64 and LP64 interfaces are out of scope of the *Intel MKL Reference Manual*, you are encouraged to browse the include files, examples, and tests for the ILP64 interface details. To do this, see the following directories, respectively:

`<mkl directory>/include`

`<mkl directory>/examples`

`<mkl directory>/tests`

This section shows

- How the ILP64 concept is implemented specifically for Intel MKL
- How to compile your code for the ILP64 interface
- How to code for the ILP64 interface
- How to browse the Intel MKL include files for the ILP64 interface

This section also explains limitations of the ILP64 support.

### Concept

ILP64 interface is provided for the following two reasons:

- To support huge data arrays (with more than 2 billion elements)
- To enable compiling your Fortran code with the `-i8` compiler option.

The Intel® Fortran Compiler supports the `-i8` option for changing behavior of the `INTEGER` type. By default the standard `INTEGER` type is 4-byte. The `-i8` option makes the compiler treat `INTEGER` constants, variables, function and subroutine parameters as 8-byte.

The ILP64 binary interface uses 8-byte integers for function parameters that define array sizes, indices, strides, etc. At the language level, that is, in the `*.f90` and `*.fi` files located in the Intel MKL include directory, such parameters are declared as `INTEGER`.

To bind your Fortran code with the ILP64 interface, you must compile your code with the `-i8` compiler option. And vice-versa, if your code is compiled with `-i8`, you can bind it only with the ILP64 interface because the LP64 binary interface requires the `INTEGER` type to be 4-byte.

Note that some Intel MKL functions and subroutines have scalar or array parameters of type `INTEGER*4` or `INTEGER(KIND=4)`, which are always 4-byte, regardless of whether the code is compiled with the `-i8` option.

For the languages of C/C++, Intel MKL provides the `MKL_INT` type as a counterpart of the `INTEGER` type for Fortran. `MKL_INT` is a macro defined as the standard C/C++ type `int` by default. However, if the `MKL_ILP64` macro is defined for the code compilation, `MKL_INT` is defined as a 64-bit integer type. To define the `MKL_ILP64` macro, you may call the compiler with the `-DMKL_ILP64` command-line option.

Intel MKL also defines the type `MKL_LONG` for maintaining ILP64 interface in the specific case of FFT interface for C/C++. The `MKL_LONG` macro is defined as the standard C/C++ type `long` by default; and if the `MKL_ILP64` macro is defined for the code compilation, `MKL_LONG` is defined as a 64-bit integer type.

**NOTE.** The type `int` is 32-bit for the Intel® C++ compiler, as well as for most of modern C/C++ compilers. The type `long` is 32- or 64-bit for the Intel® C++ and compatible compilers, depending on the particular OS.

In the Intel MKL interface for the C or C++ languages, that is, in the `*.h` header files located in the Intel MKL include directory, such function parameters as array sizes, indices, strides, etc. are declared as `MKL_INT`.

The FFT interface for C/C++ is the specific case. The header file `mkl_dfti.h` uses the `MKL_LONG` type for both explicit and implicit parameters of the interface functions. Specifically, the type of the explicit parameter *`dimension`* of the function `DftiCreateDescriptor()` is `MKL_LONG` and the type of the implicit parameter *`length`* is `MKL_LONG` for a one-dimensional transform and `MKL_LONG[]` (that is, an array of numbers having type `MKL_LONG`) for a multi-dimensional transform.

To bind your C/C++ code with the ILP64 interface, you must provide the `-DMKL_ILP64` command-line option to the compiler to enforce `MKL_INT` and `MKL_LONG` being 64-bit. And vice-versa, if your code is compiled with `-DMKL_ILP64` option, you can bind it only with the ILP64 interface because the LP64 binary interface requires `MKL_INT` to be 32-bit and `MKL_LONG` to be the standard `long` type.

Note that certain MKL functions have parameters explicitly declared as `int` or `int[]`. Such integers are always 32-bit regardless of whether the code is compiled with the `-DMKL_ILP64` option.

Table 3-2 summarizes how the Intel MKL ILP64 concept is implemented:

**Table 3-2 Intel MKL ILP64 concept**

| | **Fortran** | **C or C++** |
|---|---|---|
| The same include directory for ILP64 and LP64 interfaces | `<mkl directory>/include` | |
| Type used for parameters that are always 32-bit | `INTEGER*4` | `int` |
| Type used for parameters that are 64-bit integers for the ILP64 interface and 32-bit integers for LP64 | `INTEGER` | `MKL_INT` |
| Type used for all integer parameters of the FFT functions | `INTEGER` | `MKL_LONG` |
| Command-line option to control compiling for ILP64 | `-i8` | `-DMKL_ILP64` |

## Compiling for ILP64

The same copy of the Intel MKL include directory is used for both ILP64 and LP64 interfaces. So, the compilation for the ILP64 interface looks like this:

**Fortran:**

```
ifort -i8 -I<mkl drectory>/include …
```

**C or C++:**

```
icc -DMKL_ILP64 -I<mkl directory>/include …
```

To compile for the LP64 interface, just omit the `-i8` or `-DMKL_ILP64` option.
Notice that linking of the application compiled with the `-i8` or `-DMKL_ILP64` option to the LP64 libraries may result in unpredictable consequences and erroneous output.

Table 3-3 summarizes the compiler options:

**Table 3-3 Compiler options for the ILP64 interface**

| | **Fortran** | **C or C++** |
|---|---|---|
| Compiling for the ILP64 interface | `ifort -i8 ...` | `icc -DMKL_ILP64 ...` |
| Compiling for the LP64 interface | `ifort ...` | `icc ...` |

## Coding for ILP64

Although the `*.f90`, `*.fi`, and `*.h` files in the Intel MKL include directory were changed to meet requirements of the ILP64 interface, the LP64 interface was not changed. That is, all function parameters that were 32-bit integers still retain the 32-bit integer type, and all function parameters that were standard long integers still retain the standard `long` type. So, you do not need to change existing code if you are not using the ILP64 interface.

To migrate to ILP64 or write new code for ILP64, you need to use appropriate types for parameters of the Intel MKL functions and subroutines. For the parameters that must be 64-bit integers in ILP64, you are encouraged to use the universal integer types, namely,

- `INTEGER` for Fortran
- `MKL_INT` for C/C++
- `MKL_LONG` for the parameters of the C/C++ FFT interface.

This way you make your code universal for both ILP64 and LP64 interfaces.

You may alternatively use other 64-bit types for the integer parameters that must be 64-bit in ILP64. For example, with Intel® compilers, you may use types:

- `INTEGER(KIND=8)` for Fortran
- `long long int` for C/C++

Note that code written this way will not work for the LP64 interface. Table 3-4 summarizes usage of the integer types.

**Table 3-4 Integer types**

| | **Fortran** | **C or C++** |
|---|---|---|
| 32-bit integers | `INTEGER*4`<br>or<br>`INTEGER(KIND=4)` | `int` |
| Universal integers:<br>• 64-bit for ILP64<br>• 32-bit otherwise | `INTEGER`<br>without specifying `KIND` | `MKL_INT` |

**Table 3-4 Integer types** (continued)

| | Fortran | C or C++ |
|---|---|---|
| Universal type for the FFT interface parameters | `INTEGER` without specifying `KIND` | `MKL_LONG` |

## Browsing the Intel MKL include files

Given a function with integer parameters, the *Reference Manual* does not explain which parameters become 64-bit and which remain 32-bit for ILP64.

To find out this information, you need to browse the include files, examples, or tests. You are encouraged to start with browsing the include files, as they contain prototypes for all Intel MKL functions. Then you may see the examples and tests for better understanding of the function usage.

All include files are located in the `<mkl directory>/include` directory. Table A-2 in Appendix A shows the include files to browse.

Some function domains that support only a Fortran interface (see Table A-1), still provide header files for C/C++ in the include directory. Such `*.h` files enable using a Fortran binary interface from C/C++ code and so describe the C interface, including its ILP64 aspect.

## Limitations

Note that, not all components support the ILP64 feature. Table 3-5 shows which function domains support ILP64 interface.

**Table 3-5 ILP64 support in Intel® MKL**

| Function domain | Support for ILP64 |
|---|---|
| BLAS | Yes |
| Sparse BLAS | Yes |
| LAPACK | Yes |
| PBLAS | Yes |
| ScaLAPACK | Yes |
| VML | Yes |
| VSL | Yes |
| DSS/PARDISO* solvers | Yes |
| ISS solvers | Yes |
| Optimization (Trust-Region) solvers | Yes |
| FFT | Yes |
| FFTW | No |

**Table 3-5 ILP64 support in Intel® MKL** (continued)

| Function domain | Support for ILP64 |
|---|---|
| Cluster FFT | Yes |
| PDE support: Trigonometric Transforms | Yes |
| PDE support: Poisson Solvers | Yes |
| GMP | No |
| BLAS 95 | Yes |
| LAPACK 95 | Yes |

## Intel® MKL Versions

Intel MKL for the Linux* OS distinguishes the following versions:

- IA-32 architecture; located in the `lib/32` directory.
- Intel® 64 architecture; located in the `lib/em64t` directory.
- IA-64 architecture; located in the `lib/64` directory.

See a detailed structure of these directories in Table 3-7.

## Directory Structure in Detail

The information in the table below shows a detailed structure of the architecture-specific directories of the library. For the contents of the `doc` directory, see Contents of the Documentation Directory. For the contents of subdirectories in the `benchmarks` directory, see LINPACK and MP LINPACK Benchmarks.

**Table 3-6 Detailed directory structure**

| Directory/file | Contents |
|---|---|
| **`lib/32`**[1] | Libraries for IA-32 architecture |
| **Static Libraries** | |
| ***Interface layer*** | |
| `libmkl_gf.a` | Interface library for GNU Fortran compiler |
| `libmkl_intel.a` | Interface library for Intel® compiler |
| ***Threading layer*** | |
| `libmkl_gnu_thread.a` | Parallel drivers library supporting GNU compiler |
| `libmkl_intel_thread.a` | Parallel drivers library supporting Intel compiler |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| `libmkl_pgi_thread.a` | Parallel drivers library supporting PGI compiler |
| `libmkl_sequential.a` | Sequential drivers library |
| ***Computational layer*** | |
| `libmkl_cdft.a` | Dummy library. Contains a reference to `lib/32/libmkl_cdft_core.a` |
| `libmkl_cdft_core.a` | Cluster version of FFTs |
| `libmkl_core.a` | Kernel library for IA-32 architecture |
| `libmkl_ia32.a` | Dummy library. Contains references to Intel MKL libraries `lib/32/libmkl_intel.a`, `lib/32/libmkl_intel_thread.a`, and `lib/32/libmkl_core.a`. |
| `libmkl_lapack.a` | Dummy library. Contains references to Intel MKL libraries `lib/32/libmkl_intel.a`, `lib/32/libmkl_intel_thread.a`, and `lib/32/libmkl_core.a`. |
| `libmkl_scalapack.a` | Dummy library. Contains a reference to `lib/32/libmkl_scalapack_core.a` |
| `libmkl_scalapack_core.a` | ScaLAPACK routines |
| `libmkl_solver.a` | Iterative Sparse Solver, Trust Region Solver, and GMP routines |
| `libmkl_solver_sequential.a` | Sequential version of Iterative Sparse Solver, Trust Region Solver, and GMP routines |
| ***RTL layer*** | |
| `libguide.a` | Intel® Legacy OpenMP* run-time library for static linking |
| `libiomp5.a` | Intel® Compatibility OpenMP* run-time library for static linking |
| `libmkl_blacs.a` | BLACS routines supporting the following MPICH versions: • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2 |
| `libmkl_blacs_intelmpi.a` | BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| `libmkl_blacs_intelmpi20.a` | A soft link to `lib/32/libmkl_blacs_intelmpi.a` |
| `libmkl_blacs_openmpi.a` | BLACS routines supporting OpenMPI. |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| **Dynamic Libraries** | |
| ***Interface layer*** | |
| `libmkl_gf.so` | Interface library for GNU Fortran compiler |
| `libmkl_intel.so` | Interface library for Intel® compiler |
| ***Threading layer*** | |
| `libmkl_gnu_thread.so` | Parallel drivers library supporting GNU compiler |
| `libmkl_intel_thread.so` | Parallel drivers library supporting Intel compiler |
| `libmkl_pgi_thread.so` | Parallel drivers library supporting PGI compiler |
| `libmkl_sequential.so` | Sequential drivers library |
| ***Computational layer*** | |
| `libmkl.so` | Dummy library. Contains references to Intel MKL libraries `lib/32/libmkl_intel.so`, `lib/32/libmkl_intel_thread.so`, and `lib/32/libmkl_core.so`. |
| `libmkl_core.so` | Library dispatcher for dynamic load of processor-specific kernel library |
| `libmkl_def.so` | Default kernel library (Intel® Pentium®, Pentium® Pro, and Pentium® II processors) |
| `libmkl_lapack.so` | LAPACK and DSS/PARDISO routines and drivers |
| `libmkl_p4.so` | Pentium® 4 processor kernel library |
| `libmkl_p4m.so` | Kernel library for processors based on the Intel® Core™ microarchitecture (except Intel® Core™ Duo and Intel® Core™ Solo processors, for which `mkl_p4p.so` is intended) |
| `libmkl_p4p.so` | Kernel library for Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| `libmkl_scalapack_core.so` | ScaLAPACK routines. |
| `libmkl_vml_def.so` | VML/VSL part of default kernel for old Intel® Pentium® processors |
| `libmkl_vml_ia.so` | VML/VSL default kernel for newer Intel® architecture processors |
| `libmkl_vml_p4.so` | VML/VSL part of Pentium® 4 processor kernel |
| `libmkl_vml_p4m.so` | VML/VSL for processors based on the Intel® Core™ microarchitecture |
| `libmkl_vml_p4m2.so` | VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families |
| `libmkl_vml_p4m3.so` | Kernel library for processors based on the next generation Intel® microarchitecture (Nehalem) |
| `libmkl_vml_p4p.so` | VML/VSL for Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3) |
| ***RTL layer*** | |
| `libguide.so` | Intel® Legacy OpenMP* run-time library for dynamic linking |
| `libiomp5.so` | Intel® Compatibility OpenMP* run-time library for dynamic linking |
| `libmkl_blacs_intelmpi.so` | BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| **`lib/em64t`**[1] | Libraries for Intel® 64 architecture |
| **Static Libraries** | |
| ***Interface layer*** | |
| `libmkl_gf_ilp64.a` | ILP64 interface library for GNU Fortran compiler |
| `libmkl_gf_lp64.a` | LP64 interface library for GNU Fortran compiler |
| `libmkl_intel_ilp64.a` | ILP64 interface library for Intel compiler |
| `libmkl_intel_lp64.a` | LP64 interface library for Intel compiler |
| `libmkl_intel_sp2dp.a` | SP2DP interface library for Intel compiler |
| ***Threading layer*** | |
| `libmkl_gnu_thread.a` | Parallel drivers library supporting GNU compiler |
| `libmkl_intel_thread.a` | Parallel drivers library supporting Intel compiler |
| `libmkl_pgi_thread.a` | Parallel drivers library supporting PGI compiler |
| `libmkl_sequential.a` | Sequential drivers library |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| ***Computational layer*** | |
| `libmkl_cdft.a` | Dummy library. Contains a reference to `lib/em64t/libmkl_cdft_core.a`. |
| `libmkl_cdft_core.a` | Cluster version of FFTs |
| `libmkl_core.a` | Kernel library for Intel® 64 architecture |
| `libmkl_em64t.a` | Dummy library. Contains references to Intel MKL libraries `lib/em64t/libmkl_intel_lp64.a`, `lib/em64t/libmkl_intel_thread.a`, and `lib/em64t/libmkl_core.a`. |
| `libmkl_lapack.a` | Dummy library. Contains references to Intel MKL libraries `lib/em64t/libmkl_intel_lp64.a`, `lib/em64t/libmkl_intel_thread.a`, and `lib/em64t/libmkl_core.a`. |
| | |
| `libmkl_scalapack.a` | Dummy library. Contains a reference to `lib/em64t/libmkl_scalapack_lp64.a`. |
| `libmkl_scalapack_ilp64.a` | ScaLAPACK routines library supporting ILP64 interface |
| `libmkl_scalapack_lp64.a` | ScaLAPACK routines library supporting LP64 interface |
| `libmkl_solver.a` | A dummy library. Contains a reference to `lib/em64t/libmkl_solver_lp64.a`. |
| `libmkl_solver_ilp64.a` | Iterative Sparse Solver and GMP routine library supporting ILP64 interface |
| `libmkl_solver_ilp64_sequential.a` | Sequential version of Iterative Sparse Solver and Trust Region Solver routine library supporting ILP64 interface |
| `libmkl_solver_lp64.a` | Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting LP64 interface |
| `libmkl_solver_lp64_sequential.a` | Sequential version of Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting LP64 interface |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| ***RTL layer*** | |
| `libguide.a` | Intel® Legacy OpenMP* run-time library for static linking |
| `libiomp5.a` | Intel® Compatibility OpenMP* run-time library for static linking |
| `libmkl_blacs_ilp64.a` | ILP64 version of BLACS routines supporting the following MPICH versions:<br>• Myricom* MPICH version 1.2.5.10<br>• ANL* MPICH version 1.2.5.2 |
| `libmkl_blacs_intelmpi_ilp64.a` | ILP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| `libmkl_blacs_intelmpi_lp64.a` | LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| `libmkl_blacs_intelmpi20_ilp64.a` | A soft link to `lib/em64t/libmkl_blacs_intelmpi_ilp64.a` |
| `libmkl_blacs_intelmpi20_lp64.a` | A soft link to `lib/em64t/libmkl_blacs_intelmpi_lp64.a` |
| `libmkl_blacs_lp64.a` | LP64 version of BLACS routines supporting the following MPICH versions:<br>• Myricom* MPICH version 1.2.5.10<br>• ANL* MPICH version 1.2.5.2 |
| `libmkl_blacs_openmpi_ilp64.a` | ILP64 version of BLACS routines supporting OpenMPI. |
| `libmkl_blacs_openmpi_lp64.a` | LP64 version of BLACS routines supporting OpenMPI. |
| `libmkl_blacs_sgimpt_ilp64.a` | ILP64 version of BLACS routines supporting SGI MPT. |
| `libmkl_blacs_sgimpt_lp64.a` | LP64 version of BLACS routines supporting SGI MPT. |
| **Dynamic Libraries** | |
| ***Interface layer*** | |
| `libmkl_gf_ilp64.so` | ILP64 interface library for GNU Fortran compiler |
| `libmkl_gf_lp64.so` | LP64 interface library for GNU Fortran compiler |
| `libmkl_intel_ilp64.so` | ILP64 interface library for Intel compiler |
| `libmkl_intel_lp64.so` | LP64 interface library for Intel compiler |
| `libmkl_intel_sp2dp.so` | SP2DP interface library for Intel compiler |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| ***Threading layer*** | |
| `libmkl_gnu_thread.so` | Parallel drivers library supporting GNU compiler |
| `libmkl_intel_thread.so` | Parallel drivers library supporting Intel compiler |
| `libmkl_pgi_thread.so` | Parallel drivers library supporting PGI compiler |
| `libmkl_sequential.so` | Sequential drivers library |
| ***Computational layer*** | |
| `libmkl.so` | Dummy library. Contains references to Intel MKL libraries `lib/em64t/libmkl_intel_lp64.so`, `lib/em64t/libmkl_intel_thread.so`, and `lib/em64t/libmkl_core.so`. |
| `libmkl_core.so` | Library dispatcher for dynamic load of processor-specific kernel |
| `libmkl_def.so` | Default kernel library |
| `libmkl_mc.so` | Kernel library for processors based on the Intel® Core™ microarchitecture |
| `libmkl_mc3.so` | Kernel library for processors based on the next generation Intel® microarchitecture (Nehalem) |
| `libmkl_lapack.so` | LAPACK and DSS/PARDISO routines and drivers |
| `libmkl_scalapack_ilp64.so` | ScaLAPACK routines library supporting ILP64 interface |
| `libmkl_scalapack_lp64.so` | ScaLAPACK routines library supporting LP64 interface |
| `libmkl_vml_def.so` | VML/VSL part of default kernels |
| `libmkl_vml_mc.so` | VML/VSL for processors based on the Intel® Core™ microarchitecture |
| `libmkl_vml_mc3.so` | VML/VSL for processors based on the next generation Intel® microarchitecture (Nehalem) |
| `libmkl_vml_p4n.so` | VML/VSL for Intel® Xeon® processor using Intel® 64 architecture |
| `libmkl_vml_mc2.so` | VML/VSL for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families |
| ***RTL layer*** | |
| `libguide.so` | Intel® Legacy OpenMP* run-time library for dynamic linking |
| `libiomp5.so` | Intel® Compatibility OpenMP* run-time library for dynamic linking |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| `libmkl_intelmpi_ilp64.so` | ILP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| `libmkl_intelmpi_lp64.so` | LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| **`lib/64`**[1] | Libraries for IA-64 architecture |
| **Static Libraries** | |
| ***Interface layer*** | |
| `libmkl_intel_ilp64.a` | ILP64 interface library for Intel compiler |
| `libmkl_intel_lp64.a` | LP64 interface library for Intel compiler |
| `libmkl_intel_sp2dp.a` | SP2DP interface library for Intel compiler |
| `libmkl_gf_ilp64.a` | ILP64 interface library for GNU Fortran compiler |
| `libmkl_gf_lp64.a` | LP64 interface library for GNU Fortran compiler |
| ***Threading layer*** | |
| `libmkl_intel_thread.a` | Parallel drivers library supporting Intel compiler |
| `libmkl_gnu_thread.a` | Parallel drivers library supporting GNU compiler |
| `libmkl_sequential.a` | Sequential drivers library |
| ***Computational layer*** | |
| `libmkl_cdft.a` | Dummy library. Contains a reference to `lib/64/libmkl_cdft_core.a` |
| `libmkl_cdft_core.a` | Cluster version of FFTs |
| `libmkl_core.a` | Kernel library for IA-64 architecture |
| `libmkl_ipf.a` | Dummy library. Contains references to Intel MKL libraries `lib/64/libmkl_intel_lp64.a`, `lib/64/libmkl_intel_thread.a`, and `lib/64/libmkl_core.a`. |
| `libmkl_lapack.a` | Dummy library. Contains references to Intel MKL libraries `lib/64/libmkl_intel_lp64.a`, `lib/64/libmkl_intel_thread.a`, and `lib/64/libmkl_core.a`. |
| `libmkl_scalapack.a` | Dummy library. Contains a reference to `lib/64/libmkl_scalapack_lp64.a`. |
| `libmkl_scalapack_ilp64.a` | ScaLAPACK routines library supporting ILP64 interface |
| `libmkl_scalapack_lp64.a` | ScaLAPACK routines library supporting LP64 interface |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| `libmkl_solver.a` | Dummy library. Contains a reference to `lib/64/libmkl_solver_lp64.a`. |
| `libmkl_solver_ilp64.a` | Iterative Sparse Solver and Trust Region Solver routine library supporting ILP64 interface |
| `libmkl_solver_ilp64_sequential.a` | Sequential version of Iterative Sparse Solver and Trust Region Solver routine library supporting ILP64 interface |
| `libmkl_solver_lp64.a` | Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting LP64 interface |
| `libmkl_solver_lp64_sequential.a` | Sequential version of Iterative Sparse Solver, Trust Region Solver, and GMP routine library supporting LP64 interface |
| ***RTL layer*** | |
| `libguide.a` | Intel® Legacy OpenMP* run-time library for static linking |
| `libiomp5.a` | Intel® Compatibility OpenMP* run-time library for static linking |
| `libmkl_blacs_ilp64.a` | ILP64 version of BLACS routines supporting the following MPICH versions:<br>• Myricom* MPICH version 1.2.5.10<br>• ANL* MPICH version 1.2.5.2 |
| `libmkl_blacs_intelmpi_ilp64.a` | ILP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| `libmkl_blacs_intelmpi_lp64.a` | LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |
| `libmkl_blacs_intelmpi20_ilp64.a` | A soft link to `lib/64/libmkl_blacs_intelmpi_ilp64.a` |
| `libmkl_blacs_intelmpi20_lp64.a` | A soft link to `lib/64/libmkl_blacs_intelmpi_lp64.a` |
| `libmkl_blacs_lp64.a` | LP64 version of BLACS routines supporting the following MPICH versions:<br>• Myricom* MPICH version 1.2.5.10<br>• ANL* MPICH version 1.2.5.2 |
| `libmkl_blacs_openmpi_ilp64.a` | ILP64 version of BLACS routines supporting OpenMPI. |
| `libmkl_blacs_openmpi_lp64.a` | LP64 version of BLACS routines supporting OpenMPI. |
| `libmkl_blacs_sgimpt_ilp64.a` | ILP64 version of BLACS routines supporting SGI MPT. |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| `libmkl_blacs_sgimpt_lp64.a` | LP64 version of BLACS routines supporting SGI MPT. |
| **Dynamic Libraries** | |
| ***Interface layer*** | |
| `libmkl_gf_ilp64.so` | ILP64 interface library for GNU Fortran compiler |
| `libmkl_gf_lp64.so` | LP64 interface library for GNU Fortran compiler |
| `libmkl_intel_ilp64.so` | ILP64 interface library for Intel compiler |
| `libmkl_intel_lp64.so` | LP64 interface library for Intel compiler |
| `libmkl_intel_sp2dp.so` | SP2DP interface library for Intel compiler |
| ***Threading layer*** | |
| `libmkl_gnu_thread.so` | Parallel drivers library supporting GNU compiler |
| `libmkl_intel_thread.so` | Parallel drivers library supporting Intel compiler |
| `libmkl_sequential.so` | Sequential drivers library |
| ***Computational layer*** | |
| `libmkl.so` | Dummy library. Contains references to Intel MKL libraries `lib/64/libmkl_intel_lp64.so`, `lib/64/libmkl_intel_thread.so`, and `lib/64/libmkl_core.so`. |
| `libmkl_core.so` | Library dispatcher for dynamic load of processor-specific kernel library |
| `libmkl_i2p.so` | Kernel library for IA-64 architecture |
| `libmkl_lapack.so` | LAPACK and DSS/PARDISO routines and drivers |
| `libmkl_scalapack_ilp64.so` | ScaLAPACK routines library supporting ILP64 interface |
| `libmkl_scalapack_lp64.so` | ScaLAPACK routines library supporting LP64 interface |
| `libmkl_vml_i2p.so` | VML kernel for IA-64 architecture |
| ***RTL layer*** | |
| `libguide.so` | Intel® Legacy OpenMP* run-time library for dynamic linking |
| `libiomp5.so` | Intel® Compatibility OpenMP* run-time library for dynamic linking |
| `libmkl_blacs_intelmpi_ilp64.so` | ILP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |

**Table 3-6 Detailed directory structure** (continued)

| Directory/file | Contents |
|---|---|
| `libmkl_blacs_intelmpi_lp64.so` | LP64 version of BLACS routines supporting Intel MPI 2.0 and 3.0, and MPICH2 |

1. Additionally, a number of interface libraries may be generated as a result of respective makefile operation in the `interfaces` directory (see "Using Language-Specific Interfaces with Intel® MKL" in chapter 7).

## Dummy Libraries

Layered libraries give more flexibility to choose the appropriate combination of libraries but do not have backward compatibility by library names in link lines. Dummy libraries are introduced to provide backward compatibility with earlier version of Intel MKL, which did not use layered libraries.

Dummy libraries do not contain any functionality, but only dependencies on a set of layered libraries. Placed in a link line, dummy libraries enable omitting dependent layered libraries, which will be linked automatically. Dummy libraries contain dependency on the following layered libraries (default principle):

- Interface: Intel, LP64
- Threading: Intel compiled
- Computational: the computation library.

So, if you employ the above interface and use OpenMP* threading provided by the Intel® compiler, you may not change your link lines.

## Accessing the Intel® Math Kernel Library Documentation

This section details the contents of the Intel MKL documentation directory and explains how to access man pages for the library.

## Contents of the Documentation Directory

Table 3-7 shows the contents of the `doc` subdirectory in the Intel MKL installation directory:

**Table 3-7 Contents of the doc directory**

| File name | Comment |
|---|---|
| `Install.txt` | Intel MKL Installation Guide |
| `mkl_documentation.htm` | Index of the Intel MKL documentation |

**Table 3-7 Contents of the doc directory** (continued)

| File name | Comment |
|---|---|
| `mklEULA.txt` | Intel MKL license |
| `mklman.pdf` | Intel MKL Reference Manual |
| `mklman90_j.pdf` | Intel MKL Reference Manual in Japanese |
| `mklsupport.txt` | Information on package number for customer support reference |
| `Readme.txt` | Intel MKL Initial User Information |
| `Release_Notes.htm` | Intel MKL Release Notes (HTML format) |
| `Release_Notes.txt` | Intel MKL Release Notes (text format) |
| `userguide.pdf` | Intel MKL User's Guide, this document. |
| `vmlnotes.htm` | General discussion of VML |
| `vslnotes.pdf` | General discussion of VSL |
| `./tables` | Directory that contains tables referenced in `vmlnotes.htm`. |

## Accessing Man Pages

During installation, the man pages for the Intel MKL functions are copied to subdirectory `man/man3` of the Intel MKL installation directory.

To make the man pages accessible through the `man` command in your command shell, add the directory with the man pages to the `MANPATH` environment variable (see Setting Environment Variables in chapter 4).

Once the environment variable is set, to view the man page for an Intel MKL function, enter the following command in your command shell:

```
man <function base name>
```

In this release, *`<function base name>`* is the function name with omitted prefixes denoting data type, precision or function domain.

Examples:

- For the BLAS function `ddot`, enter `man dot`
- For the ScaLAPACK function `pzgeql2`, enter `man pgeql2`
- For the FFT function `DftiCommitDescriptor`, enter `man CommitDescriptor`.

**NOTE.** Function names in the `man` command are case-sensitive.

# Configuring Your Development Environment 4

This chapter explains how to configure your development environment for the use with the Intel® Math Kernel Library (Intel® MKL).

## Setting Environment Variables

When the installation of Intel MKL for the Linux* OS is complete, you can use three scripts `mklvars32`, `mklvarsem64t`, and `mklvars64` with two flavors each (`.sh` and `.csh`) in the `tools/environment` directory to set the environment variables `INCLUDE`, `LD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, and `FPATH` in the user shell. Section Automating the Process explains how to automate setting of these variables at startup.

For information on how to set up environment variables for threading, see Setting the Number of Threads Using OpenMP* Environment Variable in Chapter 6.

## Automating the Process

To automate setting of the environment variables `INCLUDE`, `LD_LIBRARY_PATH`, `MANPATH`, `CPATH`, `FPATH`, and `LIBRARY_PATH` at startup, add `mklvars*.sh` to your shell profile so that each time you login, the script will execute and set the path to the appropriate Intel MKL directories.

With the local user account, you should edit the following files by adding the appropriate script to the path manipulation section right before exporting variables:

- bash:

  ```
  ~/.bash_profile, ~/.bash_login or ~/.profile

  # setting up MKL environment for bash
  . <absolute_path_to_installed_MKL>/tools/environment/mklvars<arch>.sh
  ```

- sh:

```
~/.profile

# setting up MKL environment for sh
. <absolute_path_to_installed_MKL>/tools/environment/mklvars<arch>.sh
```

- csh:

```
~/.login

# setting up MKL environment for csh
. <absolute_path_to_installed_MKL>/tools/environment/mklvars<arch>.csh
```

In the above commands, `mklvars<arch>` stands for each of `mklvars32`, `mklvarsem64t` or `mklvars64`.

If you have super user permissions, you can add the same commands to a general-system file in `/etc/profile` (for bash and sh) or in `/etc/csh.login` (for csh).

Before uninstalling Intel MKL, remove the above commands from all profile files where the script execution was added, to avoid problems during logging in.

## Configuring the Eclipse* IDE CDT to Link with Intel MKL

This section describes how to configure the Eclipse* IDE C/C++ Development Tools (CDT) 3.x and 4.0 to link with Intel MKL.

**TIP.** After linking your CDT with Intel MKL, you can benefit from the Eclipse-provided *code assist* feature. See *Code/Context Assist* description in *Eclipse Help*.

### Configuring the Eclipse* IDE CDT 4.0

To configure Eclipse CDT 4.0 to link with Intel MKL, follow the instructions below:

1. If the tool-chain/compiler integration supports include path options, go to the **Includes** tab of the **C/C++ General > Paths and Symbols** property page and set the Intel MKL include path, that is, `<mkl directory>/include`.
2. If the tool-chain/compiler integration supports library path options, go to the **Library Paths** tab of the **C/C++ General > Paths and Symbols** property page and set the Intel MKL library path, depending upon the target architecture, for example, `<mkl directory>/lib/em64t`.
3. For a particular build, go to the **Tool Settings** tab of the **C/C++ Build > Settings** property page and specify names of the Intel MKL libraries to link with your application, for example, `mkl_solver_lp64` and `mkl_core` (As compilers typically require library names rather than library file names, the "`lib`" prefix and "`a`" extension are omitted). To learn how to choose the libraries, see "Selecting Libraries to Link" in chapter 5. The name of the particular setting where libraries are specified depends upon the compiler integration.

Note that the compiler/linker will automatically pick up the include and library paths settings only in cases where the automatic makefile generation is turned on, otherwise, you will have to specify the include and library paths directly in the makefile to be used.

## Configuring the Eclipse* IDE CDT 3.x

To configure Eclipse CDT 3.x to link with Intel MKL, follow the instructions below:

- For Standard Make projects,
    1. Go to **C/C++ Include Paths and Symbols** property page and set the Intel MKL include path, that is, `<mkl directory>/include`.
    2. Go to the **Libraries** tab of the **C/C++ Project Paths** property page and set the Intel MKL libraries to link with your applications, for example, `<mkl directory>/lib/em64t/libmkl_lapack.a` and `<mkl directory>/lib/em64t/libmkl_core.a`.
       To learn how to choose the libraries, see "Selecting Libraries to Link" in chapter 5.

    Note that with the Standard Make, the above settings are needed for the CDT internal functionality only. The compiler/linker will not automatically pick up these settings and you will still have to specify them directly in the makefile.
- For Managed Make projects, you can specify settings for a particular build. To do this,
    1. Go to the **Tool Settings** tab of the **C/C++ Build** property page. All the settings you need to specify are on this page. Names of the particular settings depend upon the compiler integration and therefore are not given below.
    2. If the compiler integration supports include path options, set the Intel MKL include path, that is, `<mkl_directory>/include`.

3. If the compiler integration supports library path options, set a path to the Intel MKL libraries, depending upon the target architecture, for example, `<mkl directory>/lib/em64t`.
4. Specify names of the Intel MKL libraries to link with your application, for example, `mkl_lapack` and `mkl_ia32` (As compilers typically require library names rather than library file names, the "`lib`" prefix and "`a`" extension are omitted). To learn how to choose the libraries, see "Selecting Libraries to Link" in chapter 5.

## Note on the Configuration file for Out-of-Core (OOC) DSS/PARDISO* Solver

When using the configuration file for the OOC DSS/PARDISO* Solver, mind that the maximum length of the OOC path in it is 1000.

# *Linking Your Application with Intel® Math Kernel Library* 5

This chapter discusses linking your applications with Intel® Math Kernel Library (Intel® MKL) for the Linux* OS. The chapter compares static and dynamic linking models; describes the general link line syntax to be used for linking with Intel MKL libraries; features information in a tabular form on the libraries that should be linked with your application for your particular platform and function domain; provides linking examples. Building custom shared objects is also discussed.

## Selecting Between Linkage Models

You can link your applications with Intel MKL statically, using static libraries, or dynamically, using shared libraries.

### Static Linking

Static linking resolves all symbolic references at link time. The behavior of statically built executables is predictable, because there are no run-time dependencies. The main disadvantage is that having to relink new versions of the library to your application may be error-prone and time-consuming, because you have to relink the entire application. Moreover, static linking results in large executables and uses memory less efficiently. If several executables are linked with the same library, each of them must load it into memory independently. This matters most for executables having data sizes that are small and comparable with the size of the executable.

### Dynamic Linking

Dynamic linking postpones the resolution of some undefined symbolic references until run time. Dynamically built executables contain those symbols along with a list of libraries that provide definitions of the symbols. When the executable is loaded, the final linking is done before the application runs. If several dynamically built executables reference the same library, it is loaded into memory only once and the executables share it, thereby saving memory. Dynamic linking enables you to separately update the libraries on which applications depend and does not require relinking the applications. The development advantages of dynamic linking are achieved at some cost to performance, because every unresolved symbol has to be looked up in a dedicated table and resolved at run time.

## Making the Choice

It is up to you to select whether to link in Intel MKL libraries dynamically or statically when building your application.

If you are developing just a single application and want to ship only that executable, use static linking. To reduce the size of executables shipped, you can also build custom dynamic libraries (see Building Custom Shared Objects).

Table 5-1 compares the linkage models.

**Table 5-1 Quick comparison of Intel® MKL linkage models**

| Feature | Dynamic Linkage | Static Linkage | Custom Dynamic Linkage |
|---|---|---|---|
| **Processor Updates** | Automatic | Automatic | Recompile and redistribute |
| **Optimization** | All processors | All processors | All processors |
| **Build** | Link to dynamic libraries | Link to static libraries | Build separate dynamic libraries and link to them. |
| **Calling** | Regular names | Regular names | Modified names |
| **Total Binary Size** | Large | Small | Small |
| **Executable Size** | Smallest | Small | Smallest |
| **Multi-threaded / thread-safe** | Yes[1] | Yes | Yes |

1. Except for LAPACK deprecated routines ?lacon, ?lasq3, and ?lasq4.

## Intel MKL-specific Linking Recommendations

You are strongly encouraged to dynamically link in Intel® Compatibility OpenMP* run-time library `libiomp` and Intel® Legacy OpenMP* run-time library `libguide`. Linking to static OpenMP* run-time library is not recommended because it is very easy with layered software to link in more than one copy of the library. This causes performance problems (too many threads) and may cause correctness problems if more than one copy is initialized.

You are advised to link with `libiomp` and `libguide` dynamically even if other libraries are linked statically.

## Link Command Syntax

To link libraries having filenames `libzzz.a` or `libzzz.so` with your application, two options are available:

- In the link line, list library filenames using relative or absolute paths, for example:

  ```
  <ld> myprog.o <mkl directory>/lib/32/libmkl_solver.a
  <mkl directory>/lib/32/libmkl_intel.a
  <mkl directory>/lib/32/libmkl_intel_thread.a
  <mkl directory>/lib/32/libmkl_core.a
  <mkl directory>/lib/32/libiomp5.so -lpthread
  ```

  where *<ld>* is a linker, `myprog.o` is the user's object file. Appropriate Intel MKL libraries are listed first and followed by the system library `libpthread`.
- In the link line, list library names (with absolute or relative paths, if needed) preceded with `-L`*<path>*, which indicates where to search for binaries, and `-I`*<include>*, which indicates where to search for header files. Discussion of linking with Intel MKL libraries employs this option.

To link with Intel MKL libraries, specify paths and libraries in the link line as shown below.

**NOTE.** The syntax below is provided for dynamic linking. For static linking, replace each library name preceded with "`-l`" with the path to the library file, for example, replace `-lmkl_core` with `$MKLPATH/libmkl_core.a`, where `$MKLPATH` is the appropriate user-defined environment variable. See specific examples in the Linking Examples section.

```
<files to link>
-L<MKL path> -I<MKL include>
[-lmkl_lapack95] [-lmkl_blas95]
[cluster components]
```

```
[{-lmkl_{intel, intel_ilp64, intel_lp64, intel_sp2dp, gf, gf_ilp64, gf_lp64}]
[-lmkl_{intel_thread, gnu_thread, pgi_thread, sequential}]
[{-lmkl_solver, -lmkl_solver_lp64, -lmkl_solver_ilp64}]
{{[-lmkl_lapack] -lmkl_{ia32, em64t, ipf}},
-lmkl_core}}
[{-liomp5, -lguide}] [-lpthread] [-lm]
```

See Selecting Libraries to Link for details of this syntax usage and specific recommendations on which libraries to link depending on your Intel MKL usage scenario. See also

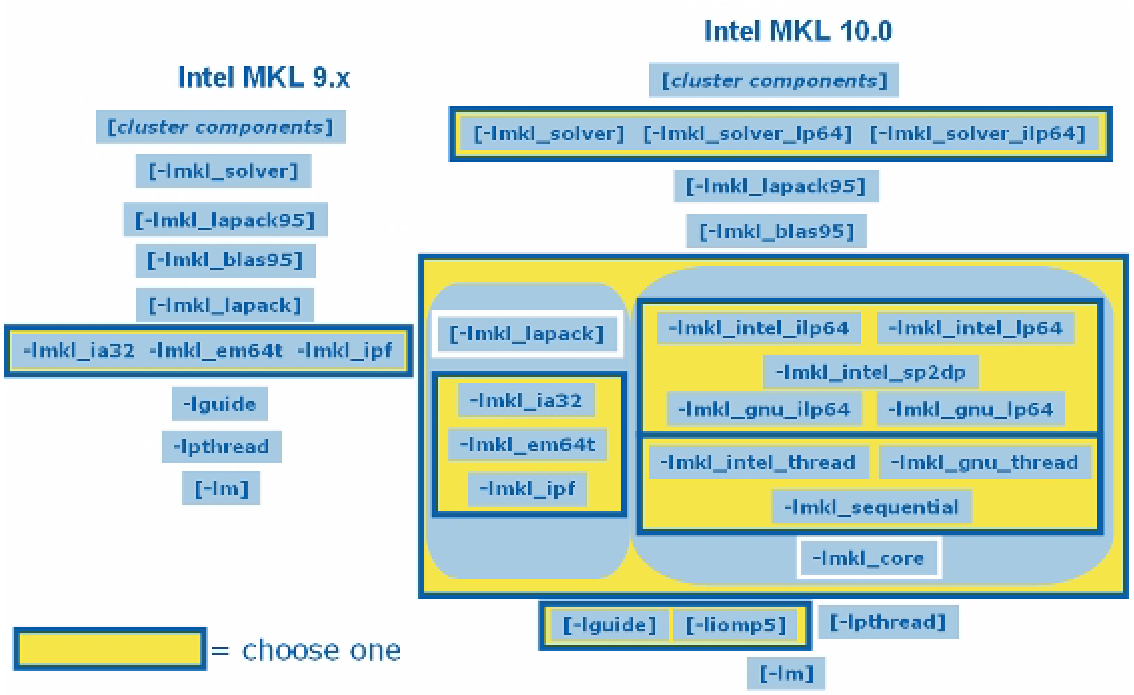
- "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" in chapter 7 for information on the libraries that you should build prior to linking
- "Working with Intel® Math Kernel Library Cluster Software" on linking with cluster components.

To link with Intel MKL, you can choose *layered model* or *legacy model*, which is backward compatible on link line (except cluster components). The syntax above incorporates both models.

For the layered model, you need to choose one library from the Interface layer, one library from the Threading layer, the Computational layer library (no choice here), and add run-time libraries. In case of the legacy model, you need not change the link line with respect to the one used with Intel MKL 9.x (see the Dummy Libraries section in chapter 3 for details).

Figure 5-1 compares linking for the Intel MKL version 10.0 or higher, which uses layers, and Intel MKL 9.x.

**Figure 5-1 Linking with Layered Intel® Math Kernel Library**



In case of employing the layered model for static linking, the cluster component, Interface layer, Threading layer, and Computational layer libraries must be enclosed in grouping symbols (for example, `-Wl,--start-group $MKLPATH/libmkl_cdft_core.a $MKLPATH/libmkl_blacs_intelmpi_ilp64.a $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group`). See specific examples in the Linking Examples section.

If you use dummy libraries,

- The path to Intel MKL libraries must be added to the list of paths that the linker will search for archive libraries (for example, as `-L<MKL path>`)
- No Interface layer or Threading layer libraries should be included in the link line
- No grouping symbols must be employed.

The order of listing libraries in the link line is essential, except for the libraries enclosed in the grouping symbols.

## Selecting Libraries to Link

Below are several simple examples of link libraries for the layered and legacy link models on 64-bit Linux* based on Intel® 64 architecture for different components using Intel® compiler interface.

- BLAS, FFT, VML, VSL components, static case:

  Legacy: `libmkl_em64t.a`

  Layered: `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`
- BLAS, FFT, VML, VSL components, dynamic case:

  Legacy: `libmkl.so`

  Layered: `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_core.so`
- LAPACK, static case:

  Legacy: `libmkl_lapack.a libmkl_em64t.a`

  Layered: `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`
- LAPACK, dynamic case:

  Legacy:`libmkl_lapack.so libmkl.so`

  Layered: `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_lapack.so libmkl_core.so`
- ScaLAPACK, static case:

  Legacy: `libmkl_scalapack.a libmkl_blacs.a libmkl_lapack.a libmkl_em64t.a`

  Layered, LP64: `libmkl_scalapack_lp64.a libmkl_blacs_lp64.a libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_lapack.a libmkl_core.a`
- ScaLAPACK, dynamic case:

  Layered, LP64: `libmkl_scalapack_lp64.so libmkl_blacs_lp64.so libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_lapack.so libmkl_core.so`
- Iterative Sparse Solver, static case:

  Legacy: `libmkl_solver.a libmkl_lapack.a libmkl_em64t.a`

  Layered, LP64: `libmkl_solver_lp64.a libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`

  Layered, ILP64: `libmkl_solver_ilp64.a libmkl_intel_ilp64.a libmkl_intel_thread.a libmkl_core.a`
- DSS/PARDISO*, static case:

  Legacy: `libmkl_lapack.a libmkl_em64t.a`

  Layered, LP64: `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`

Layered, ILP64: `libmkl_intel_ilp64.a libmkl_intel_thread.a libmkl_core.a`

- DSS/PARDISO*, dynamic case:

  Legacy: `libmkl_lapack.so libmkl_em64t.so`

  Layered, LP64: `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_lapack.so libmkl_core.so`

  Layered, ILP64: `libmkl_intel_ilp64.so libmkl_intel_thread.so libmkl_lapack.so libmkl_core.so`

When linking (see Link Command Syntax and Linking Examples), note that

- The Iterative Sparse Solver and Trust Region Solver routine library currently does not comply with the layered model. So, it is not changed internally with respect to the Intel MKL 9.x. However, to support LP64/ILP64 interfaces, two libraries were introduced in the unified structure: `libmkl_solver_lp64.a` for the LP64 interface and `libmkl_solver_ilp64.a` for the ILP64 interface. For backward link line compatibility `libmkl_solver.a` has become a dummy library. There is still only static version of the solver library, as it was for previous releases. To link with the Iterative Sparse Solver and Trust Region Solver routine library using the layered model, include the library `libmkl_solver_lp64.a` or `libmkl_solver_ilp64.a` in the link line, depending upon the interface you need.

NOTE:

**NOTE.** In MKL 10.1 Gold, the DSS/PARDISO* solver functionality was excluded from `libmkl_solver*.a` libraries and integrated into the Intel MKL layered structure. So, to use DSS/PARDISO, it is no longer necessary to link with `libmkl_solver*.a`, but the former link line is still working. Note that both static and dynamic libraries are now available for DSS/PARDISO*.

- `libmkl_lapack95.a` and `libmkl_blas95.a` libraries contain LAPACK95 and BLAS95 interfaces respectively. They are not included into the original distribution and should be built before using the interface. (See "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" and "Compiler-dependent Functions and Fortran 90 Modules" in chapter 7 for details on building the libraries and on why *source code* is distributed in this case.)
- To use the Intel MKL FFT, Trigonometric Transform, or Poisson, Laplace, and Helmholtz Solver routines, link in the math support system library by adding "`-lm`" to the link line.
- In products for Linux, it is necessary to link the `pthread` library by adding `-lpthread`. The `pthread` library is native to Linux and `libguide` makes use of this library to support multi-threading. Any time `libguide` is required, add `-lpthread` at the end of your link line (link order is important).

See also: Linking Examples
Linking with Interface Libraries
Linking with Threading Libraries
Linking with Computational Libraries

## Linking Examples

The section provides specific examples of linking supporting Intel® compilers on systems based on IA-32 and Intel® 64 architectures. In these examples, *<MKL path>* and *<MKL include>* placeholders are replaced with user-defined environment variables `$MKLPATH` and `$MKLINCLUDE`, respectively. See also examples on linking with ScaLAPACK and Cluster FFT in chapter 9.

For more linking examples, see the Intel MKL support website at http://www.intel.com/support/performancetools/libraries/mkl/.

### Linking on systems based on IA-32 architecture

1. Static linking of user's code `myprog.f` and parallel Intel MKL:

   ```
   ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
   -Wl,--start-group $MKLPATH/libmkl_intel.a
   $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
   -liomp5 -lpthread
   ```

2. Dynamic linking of user's code `myprog.f` and parallel Intel MKL:

   ```
   ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
   -lmkl_intel -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
   ```

3. Static linking of user's code `myprog.f` and sequential version of Intel MKL:

   ```
   ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
   -Wl,--start-group $MKLPATH/libmkl_intel.a
   $MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -Wl,--end-group
   -lpthread
   ```

4. Dynamic linking of user's code `myprog.f` and sequential version of Intel MKL:

   ```
   ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
   -lmkl_intel -lmkl_sequential -lmkl_core -lpthread
   ```

5. Static linking of user's code `myprog.f`, Fortran 95 LAPACK interface[1], and parallel Intel MKL:

1. See "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" in chapter 7 for information on how to build Fortran 95 LAPACK and BLAS interface libraries.

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_lapack95
-Wl,--start-group $MKLPATH/libmkl_intel.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```

6. Static linking of user's code `myprog.f`, Fortran 95 BLAS interface[1], and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_blas95
-Wl,--start-group $MKLPATH/libmkl_intel.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```

7. Static linking of user's code `myprog.f`, parallel version of an iterative sparse solver, and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver
-Wl,--start-group $MKLPATH/libmkl_intel.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```

8. Static linking of user's code `myprog.f`, sequential version of an iterative sparse solver, and sequential Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver_sequential
-Wl,--start-group $MKLPATH/libmkl_intel.a
$MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lpthread
```

## Linking on systems based on Intel® 64 architecture

1. Static linking of user's code `myprog.f` and parallel Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```

2. Dynamic linking of user's code `myprog.f` and parallel Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```

3. Static linking of user's code `myprog.f` and sequential version of Intel MKL supporting LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lpthread
```

4. Dynamic linking of user's code `myprog.f` and sequential version of Intel MKL supporting LP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread
```
5. Static linking of user's code `myprog.f` and parallel Intel MKL supporting ILP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_ilp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```
6. Dynamic linking of user's code `myprog.f` and parallel Intel MKL supporting ILP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_ilp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```
7. Static linking of user's code `myprog.f`, Fortran 95 LAPACK interface[1], and parallel Intel MKL supporting LP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_lapack95
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```
8. Static linking of user's code `myprog.f`, Fortran 95 BLAS interface[1], and parallel Intel MKL supporting LP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_blas95
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```
9. Static linking of user's code `myprog.f`, parallel version of an iterative sparse solver, and parallel Intel MKL supporting LP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver_lp64
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-liomp5 -lpthread
```
10. Static linking of user's code `myprog.f`, sequential version of an iterative sparse solver, and sequential Intel MKL supporting LP64 interface:
```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver_lp64_sequential
-Wl,--start-group $MKLPATH/libmkl_intel_lp64
$MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lpthread
```

1. See Fortran 95 Interfaces and Wrappers to LAPACK and BLAS in chapter 7 for information on how to build Fortran 95 LAPACK and BLAS interface libraries.

## Linking with Interface Libraries

### Linking with the Absoft* compilers

You can use Intel MKL with the Absoft* compilers on systems based on Intel® 64 or IA-32 architecture. Table 5-2 explains which Interface layer library must be included in the link line to link with the Absoft compilers.

**Table 5-2 Interface layer library for linking with the Absoft* compilers**

| Architecture | Programming Interface | Static Linking | Dynamic Linking |
|---|---|---|---|
| IA-32 | Does not matter | `libmkl_intel.a` | `libmkl_intel.so` |
| Intel® 64 | ILP64 | `libmkl_gf_ilp64.a` | `libmkl_gf_ilp64.so` |
| Intel® 64 | LP64 | `libmkl_gf_lp64.a` | `libmkl_gf_lp64.so` |

## Linking with Threading Libraries

Several compilers that Intel MKL supports have recently started employing the OpenMP* technology for threading. Starting with version 10.0, Intel MKL supports implementations of OpenMP that those compilers provide. If an application compiled with such a threading compiler used OpenMP threading and called threaded parts of Intel MKL versions lower than 10.0, there might be difficulties. They may arise because Intel MKL is threaded using Intel® compilers, and threading libraries from different compilers are not compatible. This can lead to performance issues, and perhaps even failures when incompatible threading is used within the same application. Starting with Intel MKL 10.0, several solutions are available in certain cases. Those solutions are provided both from the Threading Layer and the supplied run-time libraries found in the RTL Layer.

**Threading Layer.** Starting with version 10.0, Intel MKL is structured as layers. One of those layers is a Threading Layer. Because of the internal structure of the library, all of the threading represents a small amount of code. This code is compiled by different compilers (Intel, gnu and PGI compilers on Linux*) and the appropriate layer linked in with the threaded application.

**RTL Layer.** The second relevant component is the Compiler Support RTL Layer. Prior to Intel MKL 10.0, this layer included only the Intel® Legacy OpenMP* run-time compiler library `libguide`. Now you have a new choice to use the Intel® Compatibility OpenMP* run-time compiler library `libiomp`. The Compatibility library `libiomp` is an extension of `libguide` that provides support for one additional threading compiler on Linux (gnu). That is, a program threaded with a gnu compiler can safely be linked with Intel MKL and `libiomp` and execute efficiently and effectively. So, you are encouraged to use `libiomp` rather than `libguide`.

Table 5-3 shows different scenarios, depending on the threading compiler used, and the possibilities for each scenario to choose the Threading layer and RTL layer when using the current version of Intel MKL (static cases only):

**Table 5-3 Selecting the Threading Layer**

| Compiler | Application Threaded? | Threading Layer | RTL Layer Recommended | Comment |
|---|---|---|---|---|
| Intel | Does not matter | `libmkl_intel_thread.a` | `libiomp5.so` or `libguide.so` | |
| PGI | Yes | `libmkl_pgi_thread.a` or `libmkl_sequential.a` | PGI supplied | Use of `libmkl_sequential.a` removes threading from Intel MKL calls. |
| PGI | No | `libmkl_intel_thread.a` | `libiomp5.so` or `libguide.so` | |
| PGI | No | `libmkl_pgi_thread.a` | PGI supplied | |
| PGI | No | `libmkl_sequential.a` | None | |
| gnu | Yes | `libmkl_gnu_thread.a` | `libiomp5.so` or GNU OpenMP run-time library | `libiomp5` offers superior scaling performance. |
| gnu | Yes | `libmkl_sequential.a` | None | |
| gnu | No | `libmkl_intel_thread.a` | `libiomp5.so` or `libguide.so` | |
| other | Yes | `libmkl_sequential.a` | None | |
| other | No | `libmkl_intel_thread.a` | `libiomp5.so` or `libguide.so` | |

## Linking with Computational Libraries

Typically, if you are using the layered model of linking, the Computational layer contains only one library. However, certain Intel MKL function domains still require several computational link libraries. For each Intel MKL function domain, Table 5-4 lists Computational layer libraries that you must include in the link line for the layered model of linking. For more information on linking with ScaLAPACK and Cluster FFTs, see also Linking with ScaLAPACK and Cluster FFTs in chapter 9.

**Table 5-4 Computational libraries to link, by function domain**

| Function domain/ Interface | IA-32 Architecture | | Intel® 64 or IA-64 Architecture | |
|---|---|---|---|---|
| | Static | Dynamic | Static | Dynamic |
| BLAS | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| Sparse BLAS | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| BLAS95 Interface | `libmkl_blas95.a` `libmkl_core.a` | n/a[1] | `libmkl_blas95.a` `libmkl_core.a` | n/a[1] |
| CBLAS | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| LAPACK | `libmkl_core.a` | `libmkl_lapack.so` `libmkl_core.so` | `libmkl_core.a` | `libmkl_lapack.so` `libmkl_core.so` |
| LAPACK95 Interface | `libmkl_lapack95.a` `libmkl_core.a` | n/a[1] | `libmkl_lapack95.a` `libmkl_core.a` | n/a[1] |
| Iterative Sparse Solvers, Trust Region Solver, and GMP routines | `libmkl_solver.a` or `libmkl_solver_sequential.a` `libmkl_core.a` | n/a[1] | See below | n/a[1] |
| Iterative Sparse Solvers, Trust Region Solver, and GMP routines, LP64 interface | n/a[1] | n/a[1] | `libmkl_solver_lp64.a` or `libmkl_solver_lp64_sequential.a` `libmkl_core.a` | n/a[1] |
| Iterative Sparse Solvers, Trust Region Solver, and GMP routines, ILP64 interface | n/a[1] | n/a[1] | `libmkl_solver_ilp64.a` or `libmkl_solver_ilp64_sequential.a` `libmkl_core.a` | n/a[1] |

**Table 5-4 Computational libraries to link, by function domain** (continued)

| Function domain/ Interface | IA-32 Architecture | | Intel® 64 or IA-64 Architecture | |
|---|---|---|---|---|
| | **Static** | **Dynamic** | **Static** | **Dynamic** |
| Direct Sparse Solver/ PARDISO* Solver | `libmkl_core.a` | `libmkl_lapack.so` `libmkl_core.so` | `libmkl_core.a` | `libmkl_lapack.so` `libmkl_core.so` |
| Vector Math Library | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| Vector Statistical Library | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| Fourier Transform Functions | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| Trigono-metric Transform Functions | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| Poisson Library | `libmkl_core.a` | `libmkl_core.so` | `libmkl_core.a` | `libmkl_core.so` |
| ScaLAPACK[2] | `libmkl_scalapack_core.a` `libmkl_core.a` | `libmkl_scalapack_core.so` `libmkl_lapack.so` `libmkl_core.so` | See below | See below |
| ScaLAPACK, LP64 interface[2] | n/a[1] | n/a[1] | `libmkl_scalapack_lp64.a` `libmkl_core.a` | `libmkl_scalapack_lp64.so` `libmkl_lapack.so` `libmkl_core.so` |
| ScaLAPACK, ILP64 interface[2] | n/a[1] | n/a[1] | `libmkl_scalapack_ilp64.a` `libmkl_lapack.so` `libmkl_core.a` | `libmkl_scalapack_ilp64.so` `libmkl_lapack.so` `libmkl_core.so` |
| Cluster Fourier Transform Functions[2] | `libmkl_cdft_core.a` `libmkl_core.a` | n/a[1] | `libmkl_cdft_core.a` `libmkl_core.a` | n/a[1] |

1. Not applicable

2. Add also the library with BLACS routines corresponding to the used MPI. For details, see Linking with ScaLAPACK and Cluster FFTs in chapter 9.

## Notes on Linking

### Updating LD_LIBRARY_PATH

When using the Intel MKL shared libraries, do not forget to update the shared libraries environment path, that is, a system variable `LD_LIBRARY_PATH`, to include the libraries location. For example, if the Intel MKL libraries are in the *<mkl_directory>*`/lib/32` directory, the following command line can be used (assuming a bash shell):

```
export LD_LIBRARY_PATH=<mkl_directory>/lib/32:$LD_LIBRARY_PATH
```

### Linking with threading libraries

If you link with `libiomp` statically (discouraged)

- and use the Intel® compiler, then link in the `libiomp` version that comes with the compiler, that is, use `-openmp` option.
- but do not use the Intel compiler, then link in the `libiomp` version that comes with Intel MKL.

If you use dynamic linking (`libiomp5.so`) of the threading library (recommended), make sure the `LD_LIBRARY_PATH` is defined so that exactly this version of `libiomp` is found and used at run time.

## Building Custom Shared Objects

Custom shared objects enable reducing the collection of functions available in Intel MKL libraries to those required to solve your particular problems, which helps to save disk space and build your own dynamic libraries for distribution.

## Intel MKL Custom Shared Object Builder

Custom shared object builder is targeted for creation of a dynamic library (shared object) with selected functions and located in `tools/builder` directory. The builder contains a makefile and a definition file with the list of functions. The makefile has three targets: "`ia32`", "`ipf`", and "`em64t`". "`ia32`" target is used for processors using IA-32 architecture, "`ipf`" is used for IA-64 architecture, and "`em64t`" is used for Intel® Xeon® processor using Intel® 64 architecture.

## Specifying Makefile Parameters

There are several macros (parameters) for the makefile:

`export = `*`user_list`*

specifies the full name of the file that contains the list of entry point functions to be included into shared object. This file is used for definition file creation and then for export table creation. The default name is *`user_list`* (no extension).

`name = `*`mkl_custom`*

specifies the name of the created library. By default, the library `mkl_custom.so` is built.

`xerbla = `*`user_xerbla.o`*

specifies the name of the object file that contains the user's error handler. This error handler will be added to the library and used instead of the Intel MKL error handler `xerbla`. By default this parameter is not specified, and the native Intel MKL `xerbla` is used.
Note that if the user’s error handler has the same name as the Intel MKL handler, the name of the user’s handler must be upper-case, that is, XERBLA.o.

All parameters are not mandatory. In the simplest case, the command line could be `make ia32` and the values of the remaining parameters will default. As a result, `mkl_custom.so` library for processors using IA-32 architecture will be created, the functions list will be taken from the `user_list` file, and the native Intel MKL error handler `xerbla` will be used.

Another example for a more complex case is as follows:

```
make ia32 export=my_func_list.txt name=mkl_small xerbla=my_xerbla.o
```

In this case, `mkl_small.so` library for processors using IA-32 architecture will be created, the functions list will be taken from `my_func_list.txt` file, and user's error handler `my_xerbla.o` will be used.

The process is similar for processors using Intel® 64 or IA-64 architecture.

## Specifying List of Functions

Entry points in `functions_list` file should be adjusted to interface. For example, Fortran functions get an underscore character "_" as a suffix when added to the library:

```
dgemm_
ddot_
dgetrf_
```

If selected functions have several processor-specific versions, they all will be included into the custom library and managed by dispatcher.

# *Managing Performance and Memory* 6

This chapter features different ways to obtain the best performance with the Intel® Math Kernel Library (Intel® MKL): primarily, it discusses threading (see Using Intel® MKL Parallelism), then shows coding techniques and gives hardware configuration tips for improving performance. The chapter also discusses the Intel MKL memory management and shows how to redefine memory functions that the library uses by default.

## Using Intel® MKL Parallelism

Intel MKL is threaded in a number of places:

- Direct sparse solver
- LAPACK
  - — Linear equations, computational routines:
    - factorization: `*getrf`, `*gbtrf`, `*potrf`, `*pptrf`, `*sytrf`, `*hetrf`, `*sptrf`, `*hptrf`
    - solving: `*gbtrs`, `*gttrs`, `*pptrs`, `*pbtrs`, `*pttrs`, `*sytrs`, `*sptrs`, `*hptrs`, `*tptrs`, `*tbtrs`
  - — Orthogonal factorization, computational routines:
    `*geqrf`, `*ormqr`, `*unmqr`, `*ormlq`, `*unmlq`, `*ormql`, `*unmql`, `*ormrq`, `*unmrq`
  - — Singular Value Decomposition, computational routines: `*gebrd`, `*bdsqr`
  - — Symmetric Eigenvalue Problems, computational routines:
    `*sytrd`, `*hetrd`, `*sptrd`, `*hptrd`, `*steqr`, `*stedc`

  Note that a number of other LAPACK routines, which are based on threaded LAPACK or BLAS routines, make effective use of parallelism: `*gesv`, `*posv`, `*gels`, `*gesvd`, `*syev`, `*heev`, etc.
- All Level 3 BLAS, Sparse BLAS matrix-vector and matrix-matrix multiply routines for the compressed sparse row and diagonal formats
- VML

- All FFTs (except 1D transformations when DFTI_NUMBER_OF_TRANSFORMS=1 and sizes are not power of two).

NOTE

**NOTE.** For power-of-two data in 1D FFTs, Intel MKL provides parallelism for all the three supported architectures. For Intel® 64 architecture, the parallelism is provided for double complex out-of-place FFTs only.

Being designed for multi-threaded programming, Intel MKL is *thread-safe*, which means that all Intel MKL functions[1] work correctly during simultaneous execution by multiple threads. In particular, any chunk of threaded Intel MKL code provides access of multiple threads to the same shared data, while permitting only one thread at any given time to access a shared piece of data. Due to thread-safety, you can call Intel MKL from multiple threads and not worry about the function instances interfering with each other.

The library uses OpenMP* threading software, which responds to the environmental variable `OMP_NUM_THREADS` that sets the number of threads to use. Notice that there are different means to set the number of threads. In Intel MKL releases earlier than 10.0, you could use the environment variable `OMP_NUM_THREADS` (see Setting the Number of Threads Using OpenMP* Environment Variable for details) or the equivalent OpenMP run-time function calls (detailed in section Changing the Number of Threads at Run Time). Starting with version 10.0, Intel MKL also offers variables that are independent of OpenMP, such as `MKL_NUM_THREADS`, and equivalent Intel MKL functions for thread management (see Using Additional Threading Control for details). The Intel MKL variables are always inspected first, then the OpenMP variables are examined, and if neither are used, the OpenMP software chooses the default number of threads. This is a change with respect to Intel MKL versions 9.x or earlier, which used a default value of one, as the Intel® compiler OpenMP software uses the default number of threads equal to the number of processors in your system.

NOTE

**NOTE.** In Intel MKL 10.1, the OpenMP* software determines the default number of threads. The default number of threads is equal to the number of logical processors in your system for Intel OpenMP* libraries.

To achieve higher performance, you are recommended to set the number of threads to the number of real processors or physical cores. Do this by any available means, which are summarized in section Techniques to Set the Number of Threads.

1. Except LAPACK deprecated routines ?lacon, ?lasq3, and ?lasq4.

## Techniques to Set the Number of Threads

You can employ different techniques to specify the number of threads to use in Intel MKL:

- Set OpenMP or Intel MKL environment variable:
  - `OMP_NUM_THREADS`
  - `MKL_NUM_THREADS`
  - `MKL_DOMAIN_NUM_THREADS`
- Call OpenMP or Intel MKL function:
  - `omp_set_num_threads()`
  - `mkl_set_num_threads()`
  - `mkl_domain_set_num_threads()`.

When choosing the appropriate technique, take into account the following rules:

- If you employ the OpenMP techniques (`OMP_NUM_THREADS` and `omp_set_num_threads()`) only, which was the case with earlier Intel MKL versions, the library will still respond to them.
- The Intel MKL threading controls take precedence over the OpenMP techniques.
- A function call takes precedence over any environment variables. The exception is the OpenMP subroutine `omp_set_num_threads()`, which does not have precedence over Intel MKL environment variables, such as `MKL_NUM_THREADS`.
- The environment variables cannot be used to change run-time behavior in the course of the run, as they are read only once.

## Avoiding Conflicts in the Execution Environment

There are situations in which conflicts can exist in the execution environment that make the use of threads in Intel MKL problematic. They are listed here with recommendations for dealing with them. First, a brief discussion of why the problem exists is appropriate.

If the user threads the program using OpenMP directives and compiles the program with Intel® compilers, Intel MKL and the program will both use the same threading library. Intel MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads unless the user specifically requests Intel MKL to do so via the `MKL_DYNAMIC` functionality (see Using Additional Threading Control for details). However, Intel MKL can be aware that it is in a parallel region only if the threaded program and Intel MKL are using the same threading library. If the user's program is threaded by some other means, Intel MKL may operate in multithreaded mode and the performance may suffer due to overuse of the resources.

Here are several cases with recommendations depending on the threading model you employ:

**Table 6-1 How to avoid conflicts in the execution environment for your threading model**

| Threading model | Discussion |
|---|---|
| You thread the program using OS threads (`pthreads` on the Linux* OS). | If more than one thread calls the library, and the function being called is threaded, it may be important that you turn off Intel MKL threading. Set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads). |
| You thread the program using OpenMP directives and/or pragmas and compile the program using a compiler other than a compiler from Intel. | This is more problematic in that setting of `OMP_NUM_THREADS` in the environment affects both the compiler's threading library and `libiomp` (`libguide`). In this case, you should try to choose the Threading layer library that matches the layered Intel MKL with the OpenMP compiler you employ (see Linking Examples on how to do this). If this is impossible, the sequential version of Intel MKL can be used as the Threading layer. To do this, you should link with the appropriate Threading layer library: `libmkl_sequential.a` or `libmkl_sequential.so` (see the High-level Directory Structure section in chapter 3). |
| There are multiple programs running on a multiple-cpu system, as in the case of a parallelized program running using MPI for communication in which each processor is treated as a node. | The threading software will see multiple processors on the system even though each processor has a separate MPI process running on it. In this case, set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads). |

To avoid correctness and performance problems, you are also strongly encouraged to dynamically link with the Intel® Compatibility OpenMP run-time library `libiomp` and Intel® Legacy OpenMP run-time library `libguide`.

## Setting the Number of Threads Using OpenMP* Environment Variable

You can set the number of threads using the environment variable `OMP_NUM_THREADS`. To change the number of threads, in the command shell in which the program is going to run, enter:

`export OMP_NUM_THREADS=`*`<number of threads to use>`* for certain shells, such as bash.

or

`set OMP_NUM_THREADS=`*`<number of threads to use>`* for other shells, such as csh or tcsh.

See Using Additional Threading Control on how to set the number of threads using Intel MKL environment variables, for example, `MKL_NUM_THREADS`.

## Changing the Number of Threads at Run Time

It is not possible to change the number of processors during run time using the environment variables. However, you can call OpenMP API functions from your program to change the number of threads during run time. The following sample code demonstrates changing the number of threads during run time using the `omp_set_num_threads()` routine. See also Techniques to Set the Number of Threads.

To run this example, use the `omp.h` header file from the Intel® compiler package. If you do not have the Intel compiler but wish to explore the functionality in the example, use Fortran API for `omp_set_num_threads()` rather than the C version.

**Example 6-1 Changing the number of processors for threading**

```
// ******* C language *******

#include "omp.h"
#include "mkl.h"
#include <stdio.h>

#define SIZE 1000

void main(int args, char *argv[]){

   double *a, *b, *c;
   a = new double [SIZE*SIZE];
   b = new double [SIZE*SIZE];
   c = new double [SIZE*SIZE];

   double alpha=1, beta=1;
   int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0, j=0;
   char transa='n', transb='n';
   for( i=0; i<SIZE; i++){
       for( j=0; j<SIZE; j++){
           a[i*SIZE+j]= (double)(i+j);
           b[i*SIZE+j]= (double)(i*j);
           c[i*SIZE+j]= (double)0;
```

**Example 6-1 Changing the number of processors for threading** (continued)

```
            }
        }
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
        printf("row\ta\tc\n");
        for ( i=0;i<10;i++){
            printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
        }

        omp_set_num_threads(1);

        for( i=0; i<SIZE; i++){
            for( j=0; j<SIZE; j++){
                a[i*SIZE+j]= (double)(i+j);
                b[i*SIZE+j]= (double)(i*j);
                c[i*SIZE+j]= (double)0;
            }
        }
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

         printf("row\ta\tc\n");
         for ( i=0;i<10;i++){
             printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
         }
         omp_set_num_threads(2);
         for( i=0; i<SIZE; i++){
             for( j=0; j<SIZE; j++){
                 a[i*SIZE+j]= (double)(i+j);
                 b[i*SIZE+j]= (double)(i*j);
                 c[i*SIZE+j]= (double)0;
             }
         }
         cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                        m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

         printf("row\ta\tc\n");
         for ( i=0;i<10;i++){
             printf("%d:\t%f\t%f\n", i, a[i*SIZE],
     c[i*SIZE]);
         }

         delete [] a;
         delete [] b;
         delete [] c;
     }


     // ******* Fortran language *******

    PROGRAM DGEMM_DIFF_THREADS
```

**Example 6-1 Changing the number of processors for threading** (continued)

```
      INTEGER    N, I, J
      PARAMETER  (N=1000)

      REAL*8     A(N,N),B(N,N),C(N,N)
      REAL*8     ALPHA, BETA

      INTEGER*8   MKL_MALLOC
      integer   ALLOC_SIZE

      integer   NTHRS

      ALLOC_SIZE = 8*N*N
      A_PTR = MKL_MALLOC(ALLOC_SIZE,128)
      B_PTR = MKL_MALLOC(ALLOC_SIZE,128)
      C_PTR = MKL_MALLOC(ALLOC_SIZE,128)

      ALPHA = 1.1
      BETA = -1.2

      DO I=1,N
      DO J=1,N
        A(I,J) = I+J
        B(I,J) = I*j
        C(I,J) = 0.0
      END DO
      END DO

      CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)

      print *,'Row          A          C'
      DO i=1,10
        write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
      END DO

      CALL OMP_SET_NUM_THREADS(1);

      DO I=1,N
      DO J=1,N
        A(I,J) = I+J
        B(I,J) = I*j
        C(I,J) = 0.0
      END DO
      END DO

      CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)

      print *,'Row          A          C'
      DO i=1,10
        write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
      END DO
```

**Example 6-1 Changing the number of processors for threading** (continued)

```
    CALL OMP_SET_NUM_THREADS(2);

    DO I=1,N
    DO J=1,N
      A(I,J) = I+J
      B(I,J) = I*j
      C(I,J) = 0.0
    END DO
    END DO

    CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)

    print *,'Row          A          C'
    DO i=1,10
      write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
    END DO


    STOP
    END
```

## Using Additional Threading Control

Intel MKL 10.0 introduces new optional threading controls, that is, the new environment variables and service functions. They behave similar to their OpenMP equivalents, but take precedence over them. By using these controls along with OpenMP variables, you can thread the part of the application that does not call Intel MKL and the library independently from each other.

These controls enable you to specify the number of threads for Intel MKL independently of the OpenMP settings. Although Intel MKL may actually use the number of threads that differs from the one suggested, the controls will also enable you to instruct the library to try using the suggested number in the event of an undetectable threading behavior in the application calling the library.

**NOTE.** Intel MKL does not always have a choice on the number of threads for certain reasons, such as system resources.

Employing Intel MKL threading controls in your application is optional. If you do not use them, the library will mainly behave the same way as Intel MKL 9.1 in what relates to threading with the possible exception of a different default number of threads. See Note on FFT Usage for the usage differences.

Table 6-2 lists the Intel MKL environment variables for threading control, their equivalent functions, and OMP counterparts:

**Table 6-2 Intel® MKL environment variables for threading controls**

| Environment Variable | Service Function | Comment | Equivalent OMP Environment Variable |
|---|---|---|---|
| `MKL_NUM_THREADS` | `mkl_set_num_threads` | Suggests the number of threads to use. | `OMP_NUM_THREADS` |
| `MKL_DOMAIN_NUM_ THREADS` | `mkl_domain_set_num_ threads` | Suggests the number of threads for a particular function domain. | |
| `MKL_DYNAMIC` | `mkl_set_dynamic` | Enables Intel MKL to dynamically change the number of threads. | `OMP_DYNAMIC` |

**NOTE.** The functions take precedence over the respective environment variables.
In particular, if in your application, you want Intel MKL to use a given number of threads and do not want users of your application to change this via environment variables, set this number of threads by a call to `mkl_set_num_threads()`, which will have full precedence over any environment variables being set.

The example below illustrates the use of the Intel MKL function `mkl_set_num_threads()` to mimic the Intel MKL 9.x default behavior, that is, running on one thread.

**Example 6-2 Setting the number of threads to one**

```
// ******* C language *******
#include <omp.h>
#include <mkl.h>
…
mkl_set_num_threads ( 1 );
```

**Example 6-2 Setting the number of threads to one**

```
// ******* Fortran language *******
…
call mkl_set_num_threads( 1 )
```

The section further expands on the Intel MKL environment variables for threading control. See the *Intel MKL Reference Manual* for the detailed description of the threading control functions, their parameters, calling syntax, and more code examples.

## MKL_DYNAMIC

The default value of `MKL_DYNAMIC` is `TRUE`, regardless of `OMP_DYNAMIC`, whose default value may be `FALSE`.

`MKL_DYNAMIC` being `TRUE` means that Intel MKL will always try to pick what it considers the best number of threads, up to the maximum specified by the user.

`MKL_DYNAMIC` being `FALSE` means that Intel MKL will normally try not to deviate from the number of threads the user requested. Note however that setting `MKL_DYNAMIC=FALSE` does not ensure that Intel MKL will use the number of threads that you request mainly because the library may have no choice on this number for such reasons as system resources. Moreover, the library may examine the problem and pick a different number of threads than the value suggested. For example, if you attempt to do a size 1 matrix-matrix multiply across 8 threads, the library may instead choose to use only one thread because it is impractical to use 8 threads in this event.

Note also that if Intel MKL is called in a parallel region, it will use only one thread by default. If you want the library to use nested parallelism, and the thread within a parallel region is compiled with the same OpenMP compiler as Intel MKL is using, you may experiment with setting `MKL_DYNAMIC` to `FALSE` and manually increasing the number of threads.

In general, you should set `MKL_DYNAMIC` to `FALSE` only under circumstances that Intel MKL is unable to detect, for example, when nested parallelism is desired where the library is called already from a parallel section.

`MKL_DYNAMIC` being `TRUE`, in particular, provides for optimal choice of the number of threads in the following cases:

- If the requested number of threads exceeds the number of physical cores (perhaps because of hyper-threading), and `MKL_DYNAMIC` is not changed from its default value of `TRUE`, Intel MKL will scale down the number of threads to the number of physical cores.

- If you are able to detect the presence of MPI, but cannot determine if it has been called in a thread-safe mode (it is impossible to detect this with MPICH 1.2.x, for instance), and `MKL_DYNAMIC` has not been changed from its default value of `TRUE`, Intel MKL will run one thread.

## MKL_DOMAIN_NUM_THREADS

`MKL_DOMAIN_NUM_THREADS` accepts a string value *<MKL-env-string>*, which must have the following format:

*<MKL-env-string>* ::= *<MKL-domain-env-string>* { *<delimiter>* *<MKL-domain-env-string>* }

*<delimiter>* ::= [ *<space-symbol>** ] ( *<space-symbol>* | *<comma-symbol>* | *<semicolon-symbol>* | *<colon-symbol>* ) [ *<space-symbol>** ]

*<MKL-domain-env-string>* ::= *<MKL-domain-env-name>* *<uses>* *<number-of-threads>*

*<MKL-domain-env-name>* ::= MKL_ALL | MKL_BLAS | MKL_FFT | MKL_VML

*<uses>* ::= [ *<space-symbol>** ] ( *<space-symbol>* | *<equality-sign>* | *<comma-symbol>*) [ *<space-symbol>** ]

*<number-of-threads>* ::= *<positive-number>*

*<positive-number>* ::= *<decimal-positive-number>* | *<octal-number>* | *<hexadecimal-number>*

In the syntax above, `MKL_BLAS` indicates the BLAS function domain, `MKL_FFT` indicates non-cluster FFTs, and `MKL_VML` indicates the Vector Mathematics Library.

For example,

```
MKL_ALL 2 : MKL_BLAS 1 : MKL_FFT 4

MKL_ALL=2 : MKL_BLAS=1 : MKL_FFT=4

MKL_ALL=2,  MKL_BLAS=1,  MKL_FFT=4

MKL_ALL=2;  MKL_BLAS=1;  MKL_FFT=4

MKL_ALL  = 2  MKL_BLAS 1 ,  MKL_FFT  4

MKL_ALL,2: MKL_BLAS 1, MKL_FFT,4 .
```

The global variables `MKL_ALL`, `MKL_BLAS`, `MKL_FFT`, and `MKL_VML`, as well as the interface for the Intel MKL threading control functions, can be found in the `mkl.h` header file.

Table 6-3 illustrates how values of `MKL_DOMAIN_NUM_THREADS` are interpreted.

**Table 6-3 Interpretation of MKL_DOMAIN_NUM_THREADS values**

| Value of `MKL_DOMAIN_NUM_THREADS` | Interpretation |
|---|---|
| `MKL_ALL=4` | All parts of Intel MKL are suggested to try using 4 threads. The actual number of threads may be still different because of the `MKL_DYNAMIC` setting or system resource issues. The setting is equivalent to `MKL_NUM_THREADS` = 4. |
| `MKL_ALL=1, MKL_BLAS=4` | All parts of Intel MKL are suggested to use 1 thread, except for BLAS, which is suggested to try 4 threads. |
| `MKL_VML = 2` | VML is suggested to try 2 threads. The setting affects no other part of Intel MKL. |

**NOTE.** The domain-specific settings take precedence over the overall ones. For example, the "`MKL_BLAS=4`" value of `MKL_DOMAIN_NUM_THREADS` suggests to try 4 threads for BLAS, regardless of later setting `MKL_NUM_THREADS`, and a function call "`mkl_domain_set_num_threads ( 4, MKL_BLAS );`" suggests the same, regardless of later calls to `mkl_set_num_threads()`. However, pay attention to that a function call with input "`MKL_ALL`", such as "`mkl_domain_set_num_threads (4, MKL_ALL);`" is equivalent to "`mkl_set_num_threads(4)`", and thus it will be overwritten by later calls to `mkl_set_num_threads`. Similarly, the environment setting of `MKL_DOMAIN_NUM_THREADS` with "`MKL_ALL=4`" will be overwritten with `MKL_NUM_THREADS` = 2.

Whereas the `MKL_DOMAIN_NUM_THREADS` environment variable enables you set several variables at once, for example, "`MKL_BLAS=4,MKL_FFT=2`", the corresponding function does not take string syntax. So, to do the same with the function calls, you may need to make several calls, which in this example are as follows:

```
mkl_domain_set_num_threads ( 4, MKL_BLAS );

mkl_domain_set_num_threads ( 2, MKL_FFT );
```

## Setting the Environment Variables for Threading Control

To set the environment variables used for threading control, in the command shell in which the program is going to run, enter:

`export <VARIABLE NAME>=<value>` for certain shells, such as bash.

For example,

```
export MKL_NUM_THREADS=4
export MKL_DOMAIN_NUM_THREADS="MKL_ALL=1, MKL_BLAS=4"
export MKL_DYNAMIC=FALSE
```

For other shells, such as csh or tcsh, enter

```
set <VARIABLE NAME>=<value> .
```

For example,

```
set MKL_NUM_THREADS=4
set MKL_DOMAIN_NUM_THREADS="MKL_ALL=1, MKL_BLAS=4"
set MKL_DYNAMIC=FALSE
```

### Note on FFT Usage

Introduction of additional threading control made it possible to optimize the *commit* stage of the FFT implementation and get rid of double data initialization. However, this optimization requires a change in the FFT usage. Suppose you create threads in the application yourself after initializing all FFT descriptors. In this case, threading is employed for the parallel FFT computation only, the descriptors are released upon return from the parallel region, and each descriptor is used only within the corresponding thread. Starting with Intel MKL 10.0, you must explicitly instruct the library before the *commit* stage to work on one thread. To do this, set `MKL_NUM_THREADS=1` or `MKL_DOMAIN_NUM_THREADS="MKL_FFT=1"` or call the corresponding pair of service functions. Otherwise, the actual number of threads may be different because the `DftiCommitDescriptor` function is not in a parallel region. See *Example C-27a "Using Parallel Mode with Multiple Descriptors Initialized in One Thread"* in the *Intel MKL Reference Manual*.

## Tips and Techniques to Improve Performance

To obtain the best performance with Intel MKL, follow the recommendations given in the subsections below.

## Coding Techniques

To obtain the best performance with Intel MKL, ensure the following data alignment in your source code:

- arrays are aligned at 16-byte boundaries

- leading dimension values (`n*element_size`) of two-dimensional arrays are divisible by 16
- for two-dimensional arrays, leading dimension values divisible by 2048 are avoided.

## LAPACK packed routines

The routines with the names that contain the letters `HP`, `OP`, `PP`, `SP`, `TP`, `UP` in the matrix type and storage position (the second and third letters respectively) operate on the matrices in the packed format (see LAPACK "*Routine Naming Conventions*" sections in the *Intel MKL Reference Manual*). Their functionality is strictly equivalent to the functionality of the unpacked routines with the names containing the letters `HE`, `OR`, `PO`, `SY`, `TR`, `UN` in the corresponding positions, but the performance is significantly lower.

If the memory restriction is not too tight, use an unpacked routine for better performance. Note that in such a case, you need to allocate $N^2/2$ more memory than the memory required by a respective packed routine, where $N$ is the problem size (the number of equations).

For example, solving a symmetric eigenproblem with an expert driver can be speeded up through using an unpacked routine:

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w,
z, ldz, work, lwork, iwork, ifail, info),
```

where *a* is the dimension `lda-by-n`, which is at least $N^2$ elements, instead of

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info),
```

where *ap* is the dimension $N*(N+1)/2$.

## FFT functions

There are additional conditions which improve performance of the FFT functions.

**Applications based on IA-32 or Intel® 64 architecture.** The addresses of the first elements of arrays and the leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays should be divisible by cache line size, which equals

- 32 bytes for Pentium® III processor
- 64 bytes for Pentium® 4 processor
- 128 bytes for processor using Intel® 64 architecture.

**Applications based on IA-64 architecture.** The sufficient conditions are as follows:

- For the C-style FFT, the distance $L$ between arrays that represent real and imaginary parts is not divisible by 64. The best case is when $L=k*64 + 16$

- Leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays are not power of two.

## Hardware Configuration Tips

**Dual-Core Intel® Xeon® processor 5100 series systems.** To get the best Intel MKL performance on Dual-Core Intel® Xeon® processor 5100 series systems, you are advised to enable the *Hardware DPL (streaming data) Prefetcher* functionality of this processor. Configuration of this functionality is accomplished through appropriate BIOS settings where supported. Check your BIOS documentation for details.

**The use of Hyper-Threading Technology.** Hyper-Threading Technology (HT Technology) is especially effective when each thread is performing different types of operations and when there are under-utilized resources on the processor. Intel MKL fits neither of these criteria as the threaded portions of the library execute at high efficiencies using most of the available resources and perform identical operations on each thread. You may obtain higher performance when using Intel MKL *without HT Technology enabled*. See Using Intel® MKL Parallelism for information on the default number of threads, changing this number, and other relevant details.
If you run with HT enabled, performance may be especially impacted if you run on fewer threads than physical cores. Moreover, if, for example, there are two threads to every physical core, the thread scheduler may assign two threads to some cores and ignore the other ones altogether. If you are using the OpenMP* library of the Intel Compiler, read the respective User Guide on how to best set the affinity to avoid this situation. For Intel MKL, you are recommended to set `KMP_AFFINITY=granularity=fine,compact,1,0`.

## Managing Multi-core Performance

You can obtain best performance on systems with multi-core processors by requiring that threads do not migrate from core to core. To do this, bind threads to the CPU cores by setting an affinity mask to threads. You can do it using any of the following options:

- OpenMP facilities (recommended, if available), for instance, the `KMP_AFFINITY` environment variable using the Intel OpenMP library
- A system function, as in the example below.

Suppose:

- The system has two sockets with two cores each.
- 2 threads parallel application that calls the Intel MKL FFT happens to run faster than in 4 threads, but the performance in 2 threads is very unstable.

C code presented in Example 6-3 solves the problem. The code example calls the system function `sched_setaffinity` to bind the threads to the cores on different sockets. After that the Intel MKL FFT function is called.

Compile your application with the Intel compiler using the following command:

```
icc test_application.c -openmp
```

where `test_application.c` is the filename for the application.

Build the application and run it in 2 threads:

```
env OMP_NUM_THREADS=2 ./a.out
```

**Example 6-3 Setting an affinity mask by operating system means using an Intel® compiler**

```
SetThreadAffinityMask( GetCurrentThread(), mask );
#include <stdio.h>

#define __USE_GNU
// Set affinity mask
#include <sched.h>
#include <unistd.h>
#include <omp.h>

int main(void) {
  int NCPUs = sysconf(_SC_NPROCESSORS_CONF);

  printf("Using thread affinity on %i NCPUs\n", NCPUs);

#pragma omp parallel default(shared)
  {
    cpu_set_t new_mask;
    cpu_set_t was_mask;
    int tid = omp_get_thread_num();

    CPU_ZERO(&new_mask);

    // 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
    CPU_SET(tid==0 ? 0 : 2, &new_mask);

    if (sched_getaffinity(0, sizeof(was_mask), &was_mask) == -1) {
      printf("Error: sched_getaffinity(%d, sizeof(was_mask), &was_mask)\n", tid);
    }
    if (sched_setaffinity(0, sizeof(new_mask), &new_mask) == -1) {
      printf("Error: sched_setaffinity(%d, sizeof(new_mask), &new_mask)\n", tid);
    }
    printf("tid=%d new_mask=%08X was_mask=%08X\n", tid,
           *(unsigned int*)(&new_mask), *(unsigned int*)(&was_mask));
  }

  // Call Intel MKL FFT function
```

**Example 6-3 Setting an affinity mask by operating system means using an Intel® compiler** (continued)

```
   return 0;
}
```

See the *Linux Programmer's Manual* (in man pages format) for particulars of the `sched_setaffinity` function used in the above example.

## Operating on Denormals

If an Intel MKL function operates on denormals, that is, non-zero numbers that are smaller than the smallest possible non-zero number supported by a given floating-point format, or produces denormals during the computation (for instance, if the incoming data is too close to the underflow threshold), you may experience considerable performance drop. The CPU state may be set so that floating-point operations on denormals invoke the exception handler that slows down the application.

To resolve the issue, before compiling the main program, turn on the `-ftz` option, if you are using the Intel® compiler or any other compiler that can control this feature. In this case, denormals are treated as zeros at processor level and the exception handler is not invoked. Note, however, that setting this option slightly impacts the accuracy.

Another way to bring the performance back to norm is proper scaling of the input data to avoid numbers near the underflow threshold.

## FFT Optimized Radices

You can improve the performance of Intel MKL FFT if the length of your data vector permits factorization into powers of optimized radices.

In Intel MKL, the list of optimized radices depends upon the architecture:

| | |
|---|---|
| • 2, 3, 4, 5 | for IA-32 architecture |
| • 2, 3, 4, 5 | for Intel® 64 architecture |
| • 2, 3, 4, 5, 7, 11 | for IA-64 architecture. |

## Using Intel® MKL Memory Management

Intel MKL has memory management software that controls memory buffers for the use by the library functions. New buffers that the library allocates when certain functions (Level 3 BLAS or FFT) are called are not deallocated until the program ends. To get the amount of memory allocated by the memory management software, call the `MKL_MemStat()` function. If at some point your program needs to free memory, it may do so with a call to `MKL_FreeBuffers()`. If another call is made to a library function that needs a memory buffer, then the memory manager will again allocate the buffers and they will again remain allocated until either the program ends or the program deallocates the memory.

This behavior facilitates better performance. However, some tools may report the behavior as a memory leak. You can release memory in your program through the use of a function made available in Intel MKL or you can force memory releasing after each call by setting an environment variable.

The memory management software is turned on by default. To disable the software using the environment variable, set `MKL_DISABLE_FAST_MM` to any value, which will cause memory to be allocated and freed from call to call. Disabling this feature will negatively impact performance of routines such as the level 3 BLAS, especially for small problem sizes.

Using one of these methods to release memory will not necessarily stop programs from reporting memory leaks, and, in fact, may increase the number of such reports in case you make multiple calls to the library, thereby requiring new allocations with each call. Memory not released by one of the methods described will be released by the system when the program ends.

## Redefining Memory Functions

Starting with MKL 9.0, you can replace memory functions that the library uses by default with your own ones. The *memory renaming* feature enables this replacement.

### Memory renaming

In general, if users try to employ their own memory management functions instead of similar system functions (`malloc`, `free`, `calloc`, and `realloc`), actually, the memory gets managed by two independent memory management packages, which may cause memory issues. To prevent such issues, the memory renaming feature was introduced in certain Intel® libraries and in particular in Intel MKL. This feature enables users to redefine memory management functions.

Redefining is possible because Intel MKL actually uses pointers to memory functions (`i_malloc`, `i_free`, `i_calloc`, `i_realloc`) rather than the functions themselves. These pointers initially hold addresses of respective system memory management functions (`malloc`, `free`, `calloc`, `realloc`) and are visible at the application level. So, the pointer values can be redefined programmatically.

Once a user has redirected these pointers to their own respective memory management functions, the memory will be managed with user-defined functions rather than system ones. As only one (user-defined) memory management package is in operation, the issues are avoided.

Intel MKL memory management by default uses standard C run-time memory functions to allocate or free memory. These functions can be replaced using memory renaming.

## How to redefine memory functions

To redefine memory functions, you may use the following procedure:

1. Include the `i_malloc.h` header file in your code.
   (The header file contains all declarations required for an application developer to replace the memory allocation functions. This header file also describes how memory allocation can be replaced in those Intel libraries that support this feature.)
2. Redefine values of pointers `i_malloc`, `i_free`, `i_calloc`, `i_realloc` prior to the first call to MKL functions:

**Example 6-4 Redefining memory functions**

```
#include "i_malloc.h"

   . . .

   i_malloc  = my_malloc;

   i_calloc  = my_calloc;

   i_realloc = my_realloc;

   i_free    = my_free;

   . . .

// Now you may call Intel MKL functions
```

# *Language-specific Usage Options* 7

Intel® Math Kernel Library (Intel® MKL) basically provides support for Fortran and C/C++ programming. However, not all function domains support both Fortran and C interfaces (see Table A-1 in Appendix A). For example, LAPACK has no C interface. Still you can call functions comprising these domains from C using mixed-language programming.

Moreover, even if you want to use LAPACK or BLAS, which basically support Fortran, in the Fortran 95 environment, additional effort is initially required to build language-specific interface libraries and modules, being delivered as source code.

The chapter mainly focuses on mixed-language programming and the use of language-specific interfaces. It expands upon the use of Intel MKL in C language environments for function domains that basically support Fortran as well as explains usage of language-specific interfaces and, in particular, Fortran 95 interfaces to LAPACK and BLAS. In this connection, compiler-dependent functions are discussed to explain why Fortran 90 modules are supplied as sources. A separate section guides you through the process of running examples of invoking Intel MKL functions from Java.

## Using Language-Specific Interfaces with Intel® MKL

The following interface libraries and modules may be generated as a result of operation of respective makefiles located in the interfaces directory.

**Table 7-1 Interface libraries and modules**

| File name | Comment |
|---|---|
| `libmkl_blas95.a` | Contains Fortran 95 wrappers for BLAS (BLAS95) |
| `libmkl_lapack95.a` | Contains Fortran 95 wrappers for LAPACK (LAPACK95) |
| `libfftw2xc_intel.a` | Contains interfaces for FFTW version 2.x (C interface for Intel® compiler) to call Intel MKL FFTs. |
| `libfftw2xc_gnu.a` | Contains interfaces for FFTW version 2.x (C interface for GNU compiler) to call Intel MKL FFTs. |

**Table 7-1 Interface libraries and modules** (continued)

| File name | Comment |
|---|---|
| `libfftw2xf_intel.a` | Contains interfaces for FFTW version 2.x (Fortran interface for Intel compiler) to call Intel MKL FFTs. |
| `libfftw2xf_gnu.a` | Contains interfaces for FFTW version 2.x (Fortran interface for GNU compiler) to call Intel MKL FFTs. |
| `libfftw3xc_intel.a` | Contains interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL FFTs. |
| `libfftw3xc_gnu.a` | Contains interfaces for FFTW version 3.x (C interface for GNU compiler) to call Intel MKL FFTs. |
| `libfftw3xf_intel.a` | Contains interfaces for FFTW version 3.x (Fortran interface for Intel compiler) to call Intel MKL FFTs. |
| `libfftw3xf_gnu.a` | Contains interfaces for FFTW version 3.x (Fortran interface for GNU compiler) to call Intel MKL FFTs. |
| `libfftw2x_cdft_SINGLE.a` | Contains single-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs. |
| `libfftw2x_cdft_DOUBLE.a` | Contains double-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs. |
| `mkl95_blas.mod` | Contains Fortran 95 interface module for BLAS (BLAS95) |
| `mkl95_lapack.mod` | Contains Fortran 95 interface module for LAPACK (LAPACK95) |
| `mkl95_precision.mod` | Contains Fortran 95 definition of precision parameters for BLAS95 and LAPACK95 |

Section "Fortran 95 Interfaces and Wrappers to LAPACK and BLAS" shows by example how these libraries and modules are generated.

## Fortran 95 Interfaces and Wrappers to LAPACK and BLAS

Fortran 95 interfaces are provided for pure procedures and along with wrappers are delivered as sources. (For more information, see Compiler-dependent Functions and Fortran 90 Modules). The simplest way to use them is building corresponding libraries and linking them as user's libraries. To do this, you must have administrator rights. Provided the product directory is open for writing, the procedure is simple:

1. Go to the respective directory `<`*`mkl_directory`*`>/interfaces/blas95` or `<`*`mkl_directory`*`>/interfaces/lapack95`
2. Type one of the following commands:

   `make PLAT=lnx32 lib` - for IA-32 architecture

   `make PLAT=lnx32e lib` - for Intel® 64 architecture

   `make PLAT=lnx64 lib` - for IA-64 architecture.

As a result, the required library and a respective `.mod` file will be built and installed in the standard catalog of the release.

The `.mod` files can also be obtained from files of interfaces using the compiler command

`ifort -c mkl_lapack.f90` or `ifort -c mkl_blas.f90`.

These files are in the *`include`* directory.

If you do not have administrator rights, then do the following:

1. Copy the entire directory *`<mkl_directory>`*`/interfaces/blas95` or *`<mkl_directory>`*`/interfaces/lapack95`) into a user-defined directory *`<user_dir>`*
2. Copy the corresponding file (`mkl_blas.f90 or mkl_lapack.f90`) from *`<mkl_directory>`*`/include` into the user-defined directory *`<user_dir>`*`/blas95` or *`<user_dir>`*`/lapack95` respectively
3. Run one of the above `make` commands in *`<user_dir>`*`/blas95` or *`<user_dir>`*`/lapack95` with an additional variable, for instance:

   ```
   make PLAT=lnx32 INTERFACE=mkl_blas.f90 lib
   ```

   ```
   make PLAT=lnx32 INTERFACE=mkl_lapack.f90 lib
   ```

Now the required library and the `.mod` file will be built and installed in the *`<user_dir>`*`/blas95` or *`<user_dir>`*`/lapack95` directory, respectively.

By default, the ifort compiler is assumed. You may change it with an additional parameter of `make`: `FC=`*`<compiler>`*.

For instance,

```
make PLAT=lnx64 FC=<compiler> lib
```

There is also a way to use the interfaces without building the libraries.

To delete the library from the building directory, use the following commands:

| | |
|---|---|
| `make PLAT=lnx32 clean` | - for IA-32 architecture |
| `make PLAT=lnx32e clean` | - for Intel® 64 architecture |
| `make PLAT=lnx64 clean` | - for IA-64 architecture. |

## Compiler-dependent Functions and Fortran 90 Modules

Compiler-dependent functions arise whenever the compiler places into the object code function calls that are resolved in its run-time library (RTL). Linking of such code without the appropriate RTL will result in undefined symbols. MKL has been designed to minimize RTL dependencies.

Where the dependencies do arise, supporting RTL is shipped with Intel MKL. The only examples of such RTLs, except those that are relevant to the Intel MKL cluster software, are `libiomp` and `libguide`, which are the libraries for the OpenMP* code compiled with an Intel® compiler. `libiomp` and `libguide` support the threaded code in Intel MKL.

In other cases where RTL dependencies might arise, the functions are delivered as source code and it is the responsibility of the user to compile the code with whatever compiler employed.

In particular, Fortran 90 modules result in the compiler-specific code generation requiring RTL support, so, Intel MKL delivers these modules as source code.

## Mixed-language programming with Intel® MKL

Appendix A lists the programming languages supported for each Intel MKL function domain. However, you can call Intel MKL routines from different language environments. This section explains how to do this using mixed-language programming.

## Calling LAPACK, BLAS, and CBLAS Routines from C Language Environments

Not all Intel MKL function domains support both C and Fortran environments. To use Intel MKL Fortran-style functions in C/C++ environments, you should observe certain conventions, which are discussed for LAPACK and BLAS in the subsections below.

**WARNING.** Avoid calling BLAS95/LAPACK95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.

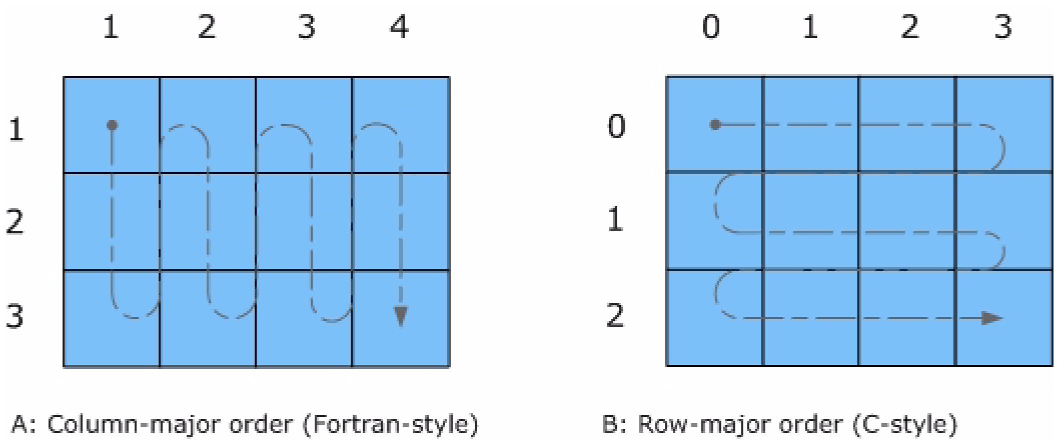
### LAPACK

As LAPACK routines are Fortran-style, when calling them from C-language programs, make sure that you follow the Fortran-style calling conventions:

- Pass variables by 'address' as opposed to pass by 'value'.
  Function calls in Example 7-2 and Example 7-3 illustrate this.
- Store your data Fortran-style, that is, in column-major rather than row-major order.

With row-major order, adopted in C, the last array index changes most quickly and the first one changes most slowly when traversing the memory segment where the array is stored. With Fortran-style column-major order, the last index changes most slowly whereas the first one changes most quickly (as illustrated by Figure 7-1 for a 2D array).

**Figure 7-1 Column-major order vs. row-major order**


1
2
3
4
1
2
3
A: Column-major order (Fortran-style)
0
1
2
3
0
1
2
B: Row-major order (C-style)

For example, if a two-dimensional matrix A of size `m x n` is stored densely in a one-dimensional array B, you can access a matrix element like this:

`A[i][j] = B[i*n+j]` in C (`i=0, ... , m-1, j=0, ... , n-1`)

`A(i,j) = B(j*m+i)` in Fortran (`i=1, ... , m, j=1, ... , n`).

When calling LAPACK routines from C, also mind that LAPACK routine names can be both upper-case or lower-case, with trailing underscore or not. For example, these names are equivalent: `dgetrf`, `DGETRF`, `dgetrf_`, `DGETRF_`.

## BLAS

BLAS routines are Fortran-style routines. If you call BLAS routines from a C-language program, you must follow the Fortran-style calling conventions:

- Pass variables by address as opposed to passing by value.
- Store data Fortran-style, that is, in column-major rather than row-major order.

Refer to the LAPACK section for details of these conventions. See Example 7-2 on how to call BLAS routines from C.

When calling BLAS routines from C, also mind that BLAS routine names can be both upper-case and lower-case, with trailing underscore or not. For example, these names are equivalent: `dgemm`, `DGEMM`, `dgemm_`, `DGEMM_`.

### CBLAS

An alternative for calling BLAS routines from a C-language program is to use the CBLAS interface.

CBLAS is a C-style interface to the BLAS routines. You can call CBLAS routines using regular C-style calls. When using the CBLAS interface, the header file `mkl.h` will simplify the program development as it specifies enumerated values as well as prototypes of all the functions. The header determines if the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. Example 7-3 illustrates the use of CBLAS interface.

## Using Complex Types in C/C++

As described in the *"Building Applications" document for the Intel® Visual Fortran Compiler 10.1*, C/C++ does not directly implement the Fortran types `COMPLEX(4)` and `COMPLEX(8)`. However, you can write equivalent structures. The type `COMPLEX(4)` has two fields each of which is a 4-byte floating-point number. The first field contains the real-number component, and the second one contains the imaginary-number component. The type `COMPLEX(8)` is similar to `COMPLEX(4)` except that each field contains an 8-byte floating-point number.

Intel MKL provides complex types `MKL_Complex8` and `MKL_Complex16`, which are structures equivalent to the Fortran complex types `COMPLEX(4)` and `COMPLEX(8)`, respectively. These types are defined in the mkl_types.h header file. You can use these types to define complex data. You can also redefine the types with your own types before including the `mkl_types.h` header file. The only requirement is that the types must be compatible with the Fortran complex layout, that is, the complex type must be a pair of real numbers for the values of real and imaginary parts.

For example, you can use the following definitions in your C++ code:

```
#define MKL_Complex8 std::complex<float>
```

and

```
#define MKL_Complex16 std::complex<double> .
```

See Example 7-2 for details. You can also define these types in the command line:

```
-DMKL_Complex8=std::complex<float>
-DMKL_Complex16=std::complex<double>.
```

## Calling BLAS Functions That Return the Complex Values in C/C++ Code

You must be careful when handling a call from C to a BLAS function that returns complex values. The problem arises because these are Fortran functions and complex return values are handled quite differently for C and Fortran. However, in addition to normal function calls, Fortran enables calling functions as though they were subroutines, which provides a mechanism for returning the complex value correctly when the function is called from a C program. When a Fortran function is called as a subroutine, the return value shows up as the first parameter in the calling sequence. This feature can be exploited by the C programmer.

The following example shows how this works.

| | |
|---|---|
| Normal Fortran function call: | `result = cdotc( n, x, 1, y, 1 ).` |
| A call to the function as a subroutine: | `call cdotc( result, n, x, 1, y, 1).` |
| A call to the function from C (notice that the hidden parameter gets exposed): | `cdotc( &result, &n, x, &one, y, &one ).` |

**NOTE.** Intel MKL has both upper-case and lower-case entry points in BLAS, with trailing underscore or not. So, all these names are acceptable: `cdotc`, `CDOTC`, `cdotc_`, `CDOTC_`.

Using the above example, you can call from C, and thus, from C++, several level 1 BLAS functions that return complex values. However, it is still easier to use the CBLAS interface. For instance, you can call the same function using the CBLAS interface as follows:

```
cblas_cdotu( n, x, 1, y, 1, &result )
```

**NOTE.** The complex value comes back expressly in this case.

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure `c`.

**Example 7-1 Calling a complex BLAS Level 1 function from C**

```
#include "mkl.h"
#define N 5
void main()
{
int n, inca = 1, incb = 1, i;
typedef struct{ double re; double im; } complex16;
complex16 a[N], b[N], c;
void zdotc();
n = N;

for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

Below is the C++ implementation:

**Example 7-2 Calling a complex BLAS Level 1 function from C++**

```
#include <complex>
#include <iostream>
#define MKL_Complex16 std::complex<double>
#include "mkl.h"

#define N 5

int main()
{
int n, inca = 1, incb = 1, i;
std::complex<double> a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){
a[i] = std::complex<double>(i,i*2.0);
b[i] = std::complex<double>(n-i,i*2.0);
}
zdotc(&c, &n, a, &inca, b, &incb );
std::cout << "The complex dot product is: " << c << std::endl;
return 0;
}
```

The implementation below uses CBLAS:

**Example 7-3 Using CBLAS interface instead of calling BLAS directly from C**

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;

extern "C" void cblas_zdotc_sub ( const int , const complex16 *,
    const int , const complex16 *, const int, const complex16*);

#define N 5

void main()
{

int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){

a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
cblas_zdotc_sub(n, a, inca, b, incb,&c );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

## Support for Boost uBLAS Matrix-Matrix Multiplication

If you have got used to uBLAS, you can perform BLAS matrix-matrix multiplication in C++ using Intel MKL substitution of Boost uBLAS functions. uBLAS pertains to the Boost C++ open-source libraries and provides BLAS functionality for dense, packed, and sparse matrices. The library uses an expression template technique for passing expressions as function arguments, which enables evaluating vector and matrix expressions in one pass without temporary matrices. uBLAS distinguishes two modes:

- Debug (safe) mode, default.
  Type and conformance checking is performed
- Release (fast) mode.
  Turned on with `NDEBUG` preprocessor symbol.

The documentation for the latest version of the Boost uBLAS is available at www.boost.org/doc/libs/1_35_0/libs/numeric/ublas/doc/index.htm.

Intel MKL provides overloaded `prod()` functions for substituting uBLAS *dense* matrix-matrix multiplication with the Intel MKL `gemm` calls. Though these functions break uBLAS expression templates and introduce temporary matrices, the performance advantage can be considerable for matrix sizes that are not too small (roughly, over 50).

You do not need to change your source code to use the functions. To call them,

- Include the header file `mkl_boost_ublas_matrix_prod.hpp` in your code
- Add appropriate Intel MKL libraries to the link line (refer to the Linking Your Application with Intel® Math Kernel Library chapter for details).

Only the following expressions are substituted:

```
prod( m1, m2 )

prod( trans(m1), m2 )

prod( trans(conj(m1)), m2 )

prod( conj(trans(m1)), m2 )

prod( m1, trans(m2) )

prod( trans(m1), trans(m2) )

prod( trans(conj(m1)), trans(m2) )

prod( conj(trans(m1)), trans(m2) )

prod( m1, trans(conj(m2)) )

prod( trans(m1), trans(conj(m2)) )

prod( trans(conj(m1)), trans(conj(m2)) )

prod( conj(trans(m1)), trans(conj(m2)) )

prod( m1, conj(trans(m2)) )

prod( trans(m1), conj(trans(m2)) )

prod( trans(conj(m1)), conj(trans(m2)) )
prod( conj(trans(m1)), conj(trans(m2)) )
```

These expressions are substituted in the *release* mode only (with NDEBUG preprocessor symbol defined). Supported uBLAS versions are Boost 1.34.1 and Boost 1.35.0. To get them, visit www.boost.org.

A code example provided in the `<mkl_directory>/examples/ublas/source/sylvester.cpp` file illustrates usage of the Intel MKL uBLAS header file for solving a special case of the Sylvester equation.

To run the Intel MKL ublas examples, specify the `BOOST_ROOT` parameter in the `make` command, for instance,

```
make lib32 BOOST_ROOT=<your_path>/boost_1_35_0
```

## Invoking Intel® MKL Functions from Java* Applications

This section describes examples that are provided with the Intel MKL package and illustrate calling the library functions from Java.

### Intel MKL Java examples

Java was positioned by its inventor, the Sun Microsystems Corporation as "Write Once Run Anywhere" (WORA) language. Intel MKL may help to speed-up Java applications, the WORA philosophy being partially supported, as Intel MKL editions are intended for wide variety of operating systems and processors covering most kinds of laptops and desktops, many workstations and servers.

To demonstrate binding with Java, Intel MKL includes the set of Java examples found in the following directory:

```
<mkl directory>/examples/java .
```

The examples are provided for the following MKL functions:

- the `?gemm`, `?gemv`, and `?dot` families from CBLAS
- complete set of non-cluster FFT functions
- ESSL[1]-like functions for 1-dimensional convolution and correlation.
- VSL Random Number Generators (RNG), except user-defined ones and file subroutines.
- VML functions, except `GetErrorCallBack`, `SetErrorCallBack`, and `ClearErrorCallBack`.

You can see the example sources in the following directory:

1. IBM Engineering Scientific Subroutine Library (ESSL*).

`<mkl directory>/examples/java/examples .`

The examples are written in Java. They demonstrate usage of the MKL functions with the following variety of data:

- 1- and 2-dimensional data sequences
- real and complex types of the data
- single and double precision.

However, note that the wrappers, used in examples, do not

- demonstrate the use of huge arrays (>2 billion elements)
- demonstrate processing of arrays in native memory
- check correctness of function parameters
- demonstrate performance optimizations

To bind with Intel MKL, the examples use the Java Native Interface (JNI* developer framework). The JNI documentation to start with is available from http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html .

The Java example set includes JNI wrappers which perform the binding. The wrappers do not depend on the examples and may be used in your Java applications. The wrappers for CBLAS, FFT, VML, VSL RNG, and ESSL-like convolution and correlation functions do not depend on each other.

To build the wrappers, just run the examples (see the Running the examples section for details). The makefile builds the wrapper binaries and the examples, invoked after that, double-check if the wrappers are built correctly. As a result of running the examples, the following directories will be created in
`<mkl directory>/examples/java`:

- `docs`
- `include`
- `classes`
- `bin`
- `_results` .

The directories `docs`, `include`, `classes`, and `bin` will contain the wrapper binaries and documentation; the directory `_results` will contain the testing results.

For a Java programmer, the wrappers look like the following Java classes:

- `com.intel.mkl.CBLAS`
- `com.intel.mkl.DFTI`
- `com.intel.mkl.ESSL`
- `com.intel.mkl.VML`

- `com.intel.mkl.VSL`

Documentation for the particular wrapper and example classes will be generated from the Java sources during building and running the examples. To browse the documentation, start from the index file in the `docs` directory which will be created by the build script:

`<`*`mkl directory`*`>/examples/java/docs/index.html .`

The Java wrappers for CBLAS, VML, VSL RNG, and FFT establish the interface that directly corresponds to the underlying native functions and you can refer to the Intel MKL Reference Manual for their functionality and parameters. Interfaces for the ESSL-like functions are described in the generated documentation for the `com.intel.mkl.ESSL` class.

Each wrapper consists of the interface part for Java and JNI stub written in C. You can find the sources in the following directory:

`<`*`mkl directory`*`>/examples/java/wrappers .`

Both Java and C parts of the wrapper for CBLAS and VML demonstrate the straightforward approach, which you may easily employ to cover additional CBLAS functions.

The wrapper for FFT is more complicated because of supporting the lifecycle for FFT descriptor objects. To compute a single Fourier transform, an application needs to call the FFT software several times with the same copy of the native FFT descriptor. The wrapper provides the handler class to hold the native descriptor while virtual machine runs Java bytecode.

The wrapper for VSL RNG is similar to the one for FFT. The wrapper provides the handler class to hold the native descriptor of the stream state.

The wrapper for the convolution and correlation functions mitigates the same difficulty of the VSL interface, which assumes similar lifecycle for "task descriptors". The wrapper utilizes the ESSL-like interface for those functions, which is simpler for the case of 1-dimensional data. The JNI stub additionally enwraps the MKL functions into the ESSL-like wrappers written in C and so "packs" the lifecycle of a task descriptor into a single call to the native method.

The wrappers meet the JNI Specification versions 1.1 and 5.0 and should work with virtually every modern implementation of Java.

The examples and the Java part of the wrappers are written for the Java language described in "*The Java Language Specification (First Edition)*" and extended with the feature of "inner classes" (this refers to late 1990s). This level of language version is supported by all versions of Sun JDK* developer toolkit and compatible implementations starting from version 1.1.5, that is, by all modern versions of Java.

The level of C language is "Standard C" (that is, C89) with additional assumptions about integer and floating-point data types required by the Intel MKL interfaces and the JNI header files. That is, the native `float` and `double` data types are required to be the same as JNI `jfloat` and `jdouble` data types, respectively, and the native `int` is required to be 4-byte long.

## Running the examples

The Java examples support all the C and C++ compilers that the Intel MKL does. The makefile intended to run the examples also needs the make utility, which is typically provided with the Linux* OS.

To run Java examples, the JDK* developer toolkit is required for compiling and running Java code. A Java implementation must be installed on the computer or available via the network. You may download the JDK from the vendor website.

The examples should work for all versions of JDK. However, they were tested only with the following Java implementations:

- J2SE* SDK 1.4.2, JDK 5.0 and 6.0 from Sun Microsystems, Inc. (http://sun.com)
- JRockit* JDK 1.4.2 and 5.0 from BEA Systems, Inc. (http://bea.com)

**NOTE.** The implementation from the Sun Microsystems Corporation supports only processors using IA-32 and Intel® 64 architectures. The implementation from BEA Systems supports Intel® Itanium® 2 processors as well.

Also note that the Java run-time environment* (JRE*) system, which may be pre-installed on your computer, is not enough. You need the JDK* developer toolkit that supports the following set of tools:

- java
- javac
- javah
- javadoc

To make these tools available for the examples makefile, set up the `JAVA_HOME` environment variable and add the JDK binaries directory to the system `PATH`, for example:

```
export JAVA_HOME=/home/<user name>/jdk1.5.0_09

export PATH=${JAVA_HOME}/bin:${PATH}
```

You may also need to clear the `JDK_HOME` environment variable, if it is assigned a value:

```
unset JDK_HOME
```

To start the examples, use the makefile found in the Intel MKL Java examples directory:

```
make {so32|soem64t|so64|lib32|libem64t|lib64} [function=…] [compiler=…]
```

If started without specifying a target (any of the choices, like `so32`), the makefile prints the help info, which explains the targets as well as the *`function`* and *`compiler`* parameters.

For the examples list, see the `examples.lst` file in the same directory.

## Known limitations

There are three kinds of limitations:

- functionality
- performance
- known bugs.

**Functionality.** It is possible that some MKL functions will not work if called from the Java environment via a wrapper, like those provided with the Intel MKL Java examples. Only those specific CBLAS, FFT, VML, VSL RNG, and the convolution/correlation functions listed in the Intel MKL Java examples section were tested with the Java environment. So, you may use the Java wrappers for these CBLAS, FFT, VML, VSL RNG, and convolution/correlation functions in your Java applications.

**Performance.** The functions from Intel MKL must work faster than similar functions written in pure Java. However, note that performance was not the main goal for these wrappers. — The intent was giving code examples. So, an Intel MKL function called from Java application will probably work slower than the same function called from a program written in C/C++ or Fortran.

**Known bugs.** There are a number of known bugs in Intel MKL (identified in the Release Notes) and there are incompatibilities between different versions of JDK. The examples and wrappers include workarounds for these problems to make the examples work anyway. Source codes of the examples and wrappers include comments which describe the workarounds.

# *Coding Tips* 8

This is another chapter whose contents discusses programming with Intel® Math Kernel Library (Intel® MKL). Whereas chapter 7 focuses on general language-specific programming options, this one presents coding tips that may be helpful to meet certain specific needs. Currently the only tip advising how to achieve numerical stability is given. You can find other coding tips, relevant to performance and memory management, in chapter 6.

## Aligning Data for Numerical Stability

If linear algebra routines (LAPACK, BLAS) are applied to inputs that are bit-for-bit identical but the arrays are differently aligned or the computations are performed either on different platforms or with different numbers of threads, the outputs may not be bit-for-bit identical, though they will deviate within the appropriate error bounds. The Intel MKL version may also affect numerical stability of the output, as the routines may be implemented differently in different versions. With a given Intel MKL version, the outputs will be bit-for-bit identical provided all the following conditions are met:

- the outputs are obtained on the same platform;
- the inputs are bit-for-bit identical;
- the input arrays are aligned identically at 16-byte boundaries.

Unlike the first two conditions, which are under users' control, the alignment of arrays, by default, is not. For instance, arrays dynamically allocated using `malloc` are aligned at 8-byte boundaries, but not at 16-byte. If you need the numerically stable output, use `MKL_malloc()` to get the properly aligned workspace:

**Example 8-1 Aligning addresses at 16-byte boundaries**

```
// ******* C language *******
...
#include <stdlib.h>
...
void *darray;
int workspace
...
// Allocate workspace aligned on 16-bit boundary
darray = MKL_malloc( sizeof(double)*workspace, 16 );
...
// call the program using MKL
mkl_app( darray );
...
// Free workspace
MKL_free( darray )

! ******* Fortran language *******
...
double precision darray
pointer (p_wrk,darray(1))
integer workspace
...
! Allocate workspace aligned on 16-bit boundary
p_wrk = mkl_malloc( 8*workspace, 16 )
...
! call the program using MKL
call mkl_app( darray )
...
! Free workspace
call mkl_free(p_wrk)
```

# Working with Intel® Math Kernel Library Cluster Software

# 9

This chapter discusses usage of Intel® MKL ScaLAPACK and Cluster FFTs, mainly describing linking your application with these domains and including C- and Fortran-specific linking examples; gives information on the supported MPI.

See Table 3-7 for detailed Intel MKL directory structure in chapter 3.

For information on the available documentation and the `doc` directory, see Table 3-7 in the same chapter.

For information on MP LINPACK Benchmark for Clusters, see section Intel® Optimized MP LINPACK Benchmark for Clusters in chapter 11.

Intel MKL ScaLAPACK and FFTs support MPICH-1.2.x and Intel® MPI.

To link a program that calls ScaLAPACK, you need to know how to link an MPI application first.

Typically, this involves using *mpi* scripts *mpicc* or *mpif77* (C or FORTRAN 77 scripts) that use the correct MPI header files and others. If, for instance, you are using MPICH installed in `/opt/mpich`, then typically `/opt/mpich/bin/mpicc` and `/opt/mpich/bin/mpif77` will be the compiler scripts and `/opt/mpich/lib/libmpich.a` will be the library used for that installation.

## Linking with ScaLAPACK and Cluster FFTs

To link to ScaLAPACK and/or Cluster FFTs in Intel MKL, basically use the following general form:

```
<<MPI> linker script> <files to link>                              \
       -L<MKL path> [-Wl,--start-group] <MKL Cluster Library>  \
       <BLACS> <MKL Core Libraries> [-Wl,--end-group]
```

where

*`<MPI>`* is one of several MPI implementations (MPICH, Intel MPI 2.x, Intel MPI 3.x)

`<BLACS>` is one of BLACS libraries for the appropriate architecture, which are listed in Table 3-6, for example, for IA-32 architecture, it is one of `-lmkl_blacs`, `-lmkl_blacs_intelmpi`, and `-lmkl_blacs_openmpi`;

`<MKL Cluster Library>` is one of ScaLAPACK or Cluster FFT libraries for the appropriate architecture, which are listed in Table 3-6, for example, for IA-32 architecture, it is one of `-lmkl_scalapack_core` or `-lmkl_cdft_core`;

`<MKL Core Libraries>` is `<MKL LAPACK & MKL kernel libraries>` for ScaLAPACK, and `<MKL kernel libraries>` for Cluster FFTs;

`<MKL kernel libraries>` are processor optimized kernels, threading library, and system library for threading support linked as described at the beginning of section Link Command Syntax in Chapter 5;

`<MKL LAPACK & kernel libraries>` are the LAPACK library and `<MKL kernel libraries>`;

grouping symbols `-Wl,--start-group` and `-Wl,--end-group` are required in case of static linking.

For example, if you are using Intel MPI 3.x, wish to statically use the LP64 interface with ScaLAPACK and to have only one MPI process per core (and thus do not employ threading), provide the following linker options:

```
-L$MKLPATH -I$MKLINCLUDE -Wl,--start-group
$MKLPATH/libmkl_scalapack_lp64.a $MKLPATH/libmkl_blacs_intelmpi_lp64.a
$MKLPATH/libmkl_intel_lp64.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a -static_mpi -Wl,--end-group -lpthread –lm
```

For more examples, see Examples for Linking with ScaLAPACK and Cluster FFT.

Note that `<<MPI> linker script>` and `<BLACS>` library should correspond to the MPI version. For instance, if it is Intel MPI 2.x, then `<Intel MPI 2.x linker script>` and `libmkl_blacs_intelmpi` libraries are used. To link with Intel MPI 3.0 or 3.1, `libmkl_blacs_intelmpi` should also be used.

For information on linking with Intel® MKL libraries, see Chapter 5 Linking Your Application with Intel® Math Kernel Library.

## Setting the Number of Threads

The OpenMP* software responds to the environmental variable `OMP_NUM_THREADS`. Intel® MKL 10.0 has also introduced other mechanisms to set the number of threads, such as `MKL_NUM_THREADS` or `MKL_DOMAIN_NUM_THREADS` (see section "Using Additional Threading Control" in chapter 6). Make certain that the relevant environment variable has the same and correct value on all the nodes. Intel MKL version 10.0 and higher also no longer sets the default number of threads to 1, but depends on the compiler to set the default number. For the threading layer based on the Intel® compiler (`libmkl_intel_thread.a`), this value is the number of CPUs according to the OS. Be cautious to avoid over-prescribing the number of threads, which may occur, for instance, when the number of MPI ranks per node and the number of threads per node are both greater than one.

The best way to set, for example, the environment variable `OMP_NUM_THREADS` is in the login environment. Remember that `mpirun` starts a fresh default shell on all of the nodes and so, changing this value on the head node and then doing the run (which works on an SMP system) will not effectively change the variable as far as your program is concerned. In `.bashrc`, you could add a line at the top, which looks like this:

```
OMP_NUM_THREADS=1; export OMP_NUM_THREADS
```

It is possible to run multiple CPUs per node using MPICH, but the MPICH must be built to allow it. Be aware that certain MPICH applications may not work perfectly in a threaded environment (see the Known Limitations section in the *Release Notes*). The safest thing for multiple CPUs, although not necessarily the fastest, is to run one MPI process per processor with `OMP_NUM_THREADS` set to one. Always verify that the combination with `OMP_NUM_THREADS=1` works correctly.

## Using Shared Libraries

All needed shared libraries must be visible on all the nodes at run time. One way to accomplish this is to point these libraries by the `LD_LIBRARY_PATH` environment variable in the `.bashrc` file. If Intel MKL is installed only on one node, you should link statically when building your Intel MKL applications.

The Intel® compilers or GNU compilers can be used to compile a program that uses Intel MKL. However, make certain that MPI implementation and compiler match up correctly.

## ScaLAPACK Tests

To build NetLib ScaLAPACK tests,

- for IA-32 architecture, add `libmkl_scalapack_core.a` to your link command
- for IA-64 and Intel® 64 architectures, add `libmkl_scalapack_lp64.a` or `libmkl_scalapack_ilp64.a`, depending upon the desired interface.

## Examples for Linking with ScaLAPACK and Cluster FFT

For information on detailed MKL structure of the architecture-specific directories of the cluster libraries, see section Directory Structure in Detail in Chapter 3.

## Examples for C Module

Suppose the following conditions are met:

- MPICH2 1.0.7 or higher is installed in `/opt/mpich`
- `$MKLPATH` is a user-defined variable containing `<mkl_directory>/lib/32`
- You use the Intel® C++ Compiler 9.1 or higher and the main module is in C.

**To link with ScaLAPACK for a cluster of systems based on the IA-32 architecture,** use the following libraries:

```
/opt/mpich/bin/mpicc <user files to link>                     \
     -L$MKLPATH                                               \
     -lmkl_scalapack_core                                     \
     -lmkl_blacs_intelmpi                                     \
     –lmkl_lapack                                             \
     -lmkl_intel –lmkl_intel_thread -lmkl_lapack –lmkl_core   \
     -liomp5 -lpthread
```

**To link with Cluster FFT for a cluster of systems based on the IA-32 architecture,** use the following libraries:

```
/opt/mpich/bin/mpicc <user files to link>      \
     $MKLPATH/libmkl_cdft_core.a               \
     $MKLPATH/libmkl_blacs_intelmpi.a          \
     $MKLPATH/libmkl_intel.a                   \
     $MKLPATH/libmkl_intel_thread.a            \
     $MKLPATH/libmkl_core.a                    \
     -liomp5 -lpthread
```

## Examples for Fortran Module

Suppose the following conditions are met:

- Intel MPI 3.0 is installed in `/opt/intel/mpi/3.0`
- `$MKLPATH` is a user-defined variable containing `<mkl_directory>/lib/64`.
- You use the Intel® Fortran Compiler 9.1 or higher and the main module is in Fortran.

**To link with ScaLAPACK for a cluster of systems based on the IA-64 architecture,** use the following libraries:

```
/opt/intel/mpi/3.0/bin/mpiifort <user files to link>          \
     -L$MKLPATH                                               \
     -lmkl_scalapack_lp64                                     \
     -lmkl_blacs_intelmpi_lp64                                \
     -lmkl_lapack                                                \
     -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_lapack -lmkl_core  \
     -liomp5 -lpthread
```

**To link with Cluster FFT for a cluster of systems based on the IA-64 architecture,** use the following libraries:

```
/opt/intel/mpi/3.0/bin/mpiifort <user files to link>        \
     $MKLPATH/libmkl_cdft_core.a                            \
     $MKLPATH/libmkl_blacs_intelmpi_ilp64.a                 \
     $MKLPATH/libmkl_intel_ilp64.a                          \
     $MKLPATH/libmkl_intel_thread.a                         \
     $MKLPATH/libmkl_core.a                                 \
     -liomp5 -lpthread
```

A binary linked with ScaLAPACK runs in the same way as any other MPI application (For information, refer to the documentation that comes with the MPI implementation). For instance, the script `mpirun` is used in case of MPICH2 and OpenMPI, and a number of MPI processes is set by `-np`. In case of MPICH 2.0 and all Intel MPIs, you should start the daemon before running an application; the execution is driven by the script `mpiexec`.

For further linking examples, see the Intel MKL support website at http://www.intel.com/support/performancetools/libraries/mkl/.

# *Getting Assistance for Programming in the Eclipse* IDE*

10

This chapter discusses features of the Intel® Math Kernel Library (Intel® MKL) which software engineers can benefit from when working in the Eclipse* IDE.

The following features assist programming in the Eclipse IDE:

- The Intel MKL Reference Manual viewable from within the IDE
- Eclipse Help search tuned to target the Intel Web sites
- Context-sensitive help in the Eclipse C/C++ Development Tools (CDT)
- Code/Content Assist in the Eclipse CDT

The first three features are provided through the Intel MKL plugin for Eclipse Help (See Table 3-1 in chapter 3 for the plugin location after installation.) To use the plugin, place it into the plugins folder of your Eclipse directory.

The last feature is native to the Eclipse CDT.

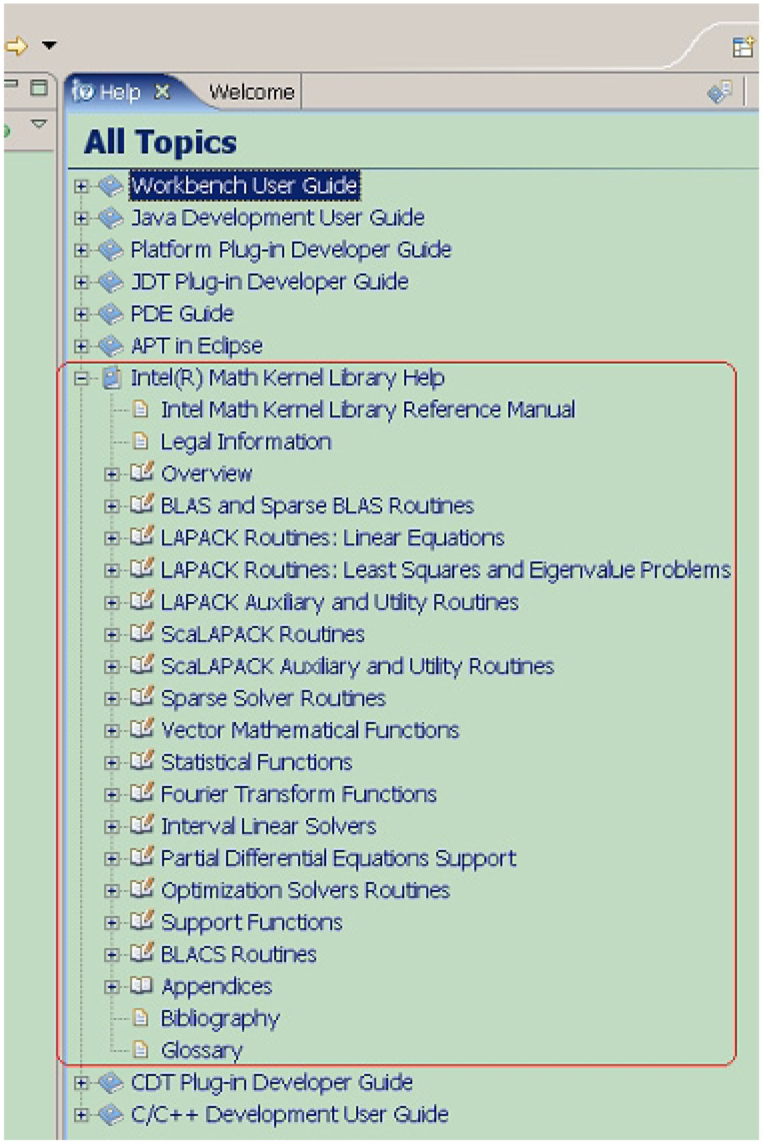
## Viewing the Intel® MKL Reference Manual in the Eclipse* IDE

To view the Reference Manual, in Eclipse,

1. Select **Help > Help Contents** from the menu
2. In the **Help** tab, click **Intel(R) Math Kernel Library Help** under **All Topics**
3. In the Help tree that expands, click **Intel Math Kernel Library Reference Manual (see** Figure 10-1**)** .

The Intel MKL Help Index is also available in Eclipse, and the Reference Manual is included in the Eclipse Help search.

**Figure 10-1 Intel® Math Kernel Library Help in the Eclipse* IDE**



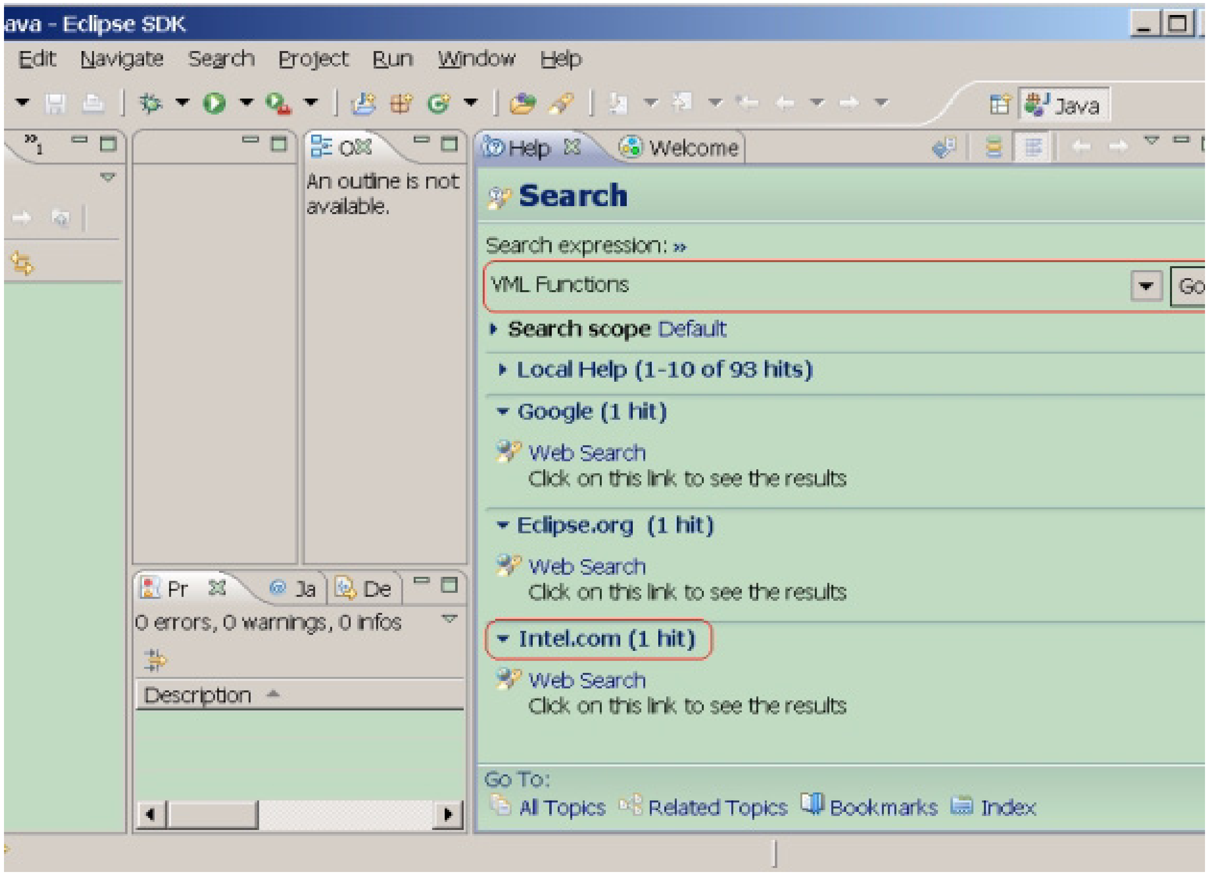
All Topics

- Workbench User Guide
- Java Development User Guide
- Platform Plug-in Developer Guide
- JDT Plug-in Developer Guide
- PDE Guide
- APT in Eclipse
- Intel(R) Math Kernel Library Help
  - Intel Math Kernel Library Reference Manual
  - Legal Information
  - Overview
  - BLAS and Sparse BLAS Routines
  - LAPACK Routines: Linear Equations
  - LAPACK Routines: Least Squares and Eigenvalue Problems
  - LAPACK Auxiliary and Utility Routines
  - ScaLAPACK Routines
  - ScaLAPACK Auxiliary and Utility Routines
  - Sparse Solver Routines
  - Vector Mathematical Functions
  - Statistical Functions
  - Fourier Transform Functions
  - Interval Linear Solvers
  - Partial Differential Equations Support
  - Optimization Solvers Routines
  - Support Functions
  - BLACS Routines
  - Appendices
  - Bibliography
  - Glossary
- CDT Plug-in Developer Guide
- C/C++ Development User Guide

## Searching the Intel Web Site from the Eclipse* IDE

The Intel MKL plugin tunes Eclipse Help search to target http://www.intel.com so that when you run a search from the Eclipse Help pane, the search hits at the site are shown through a separate link. Figure 10-2 shows search results for "VML Functions" in Eclipse Help. **1 hit** means an entry hit to the respective site.
Click "**Intel.com (1 hit)**" to open the list of actual hits to the Intel Web site.

**Figure 10-2 Hits to the Intel Web Site in the Eclipse* IDE Help search**



An outline is not available.

0 errors, 0 warnings, 0 infos

Search

Search expression: »

VML Functions

Search scope Default

Local Help (1-10 of 93 hits)

Google (1 hit)

Web Search
Click on this link to see the results

Eclipse.org (1 hit)

Web Search
Click on this link to see the results

Intel.com (1 hit)

Web Search
Click on this link to see the results

Go To:

## Using Context-Sensitive Help in the Eclipse* IDE CDT

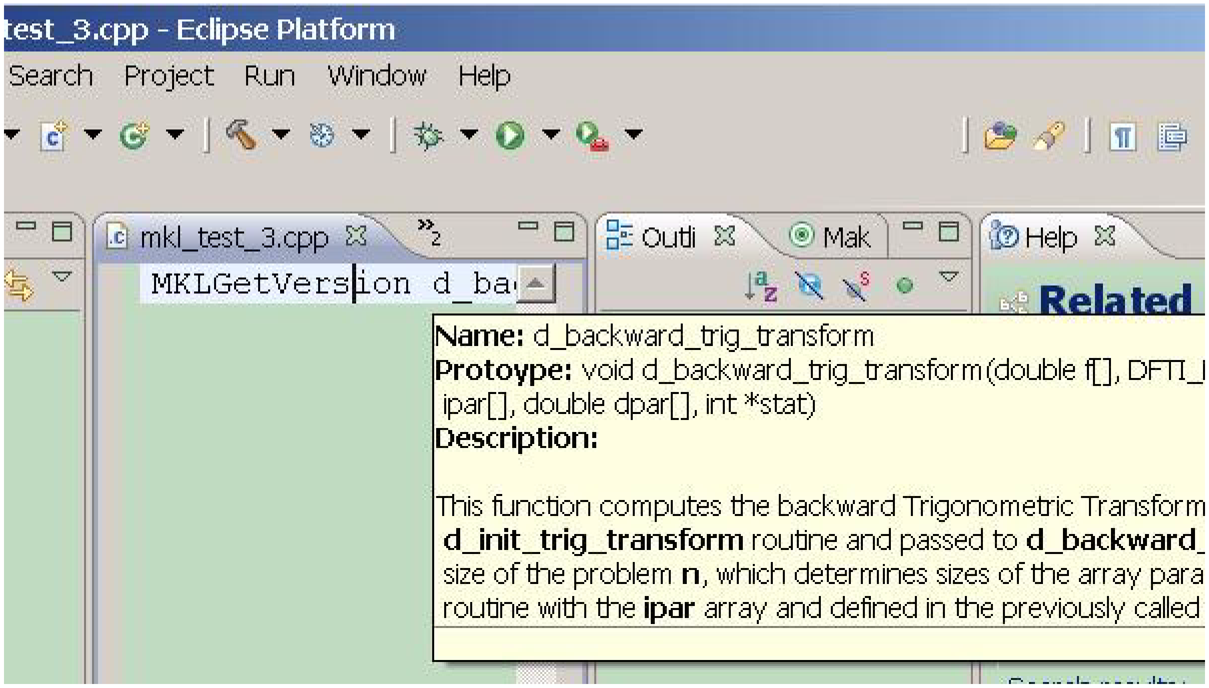
You can get context-sensitive help in the Eclipse CDT editor using Infopop windows and F1 Help.

### Infopop window

Infopop window is a popup description of a C function.

To obtain the description of an Intel MKL function whose name is typed in the editor, place the cursor over the function name.

**Figure 10-3 Infopop Window with an Intel® MKL function description**



```
MKLGetVersion d_ba
```

**Name:** d_backward_trig_transform
**Protoype:** void d_backward_trig_transform(double f[], DFTI_
ipar[], double dpar[], int *stat)
**Description:**

This function computes the backward Trigonometric Transform
**d_init_trig_transform** routine and passed to **d_backward_**
size of the problem **n**, which determines sizes of the array para
routine with the **ipar** array and defined in the previously called

### F1 Help

F1 Help basically displays the list of relevant documentation topics for a keyword.

To get F1 Help for an Intel MKL function whose name is typed in the editor window,

1. Place the cursor to the function name
2. Press F1.

   This causes two lists to display:

   — The list of links to the relevant topics in the product documentation displays in the **Related Topics** page under **See also**. The Intel MK Help Index establishes the relevance (see Figure 10-4). Typically, one link displays in this list for each function.

   — The list of search results for the function name displays in the **Related Topics** page under **Dynamic Help** (see Figure 10-5).
3. Click a needed link to open the Help topic.

**Figure 10-4 F1 Help in the Eclipse* IDE**

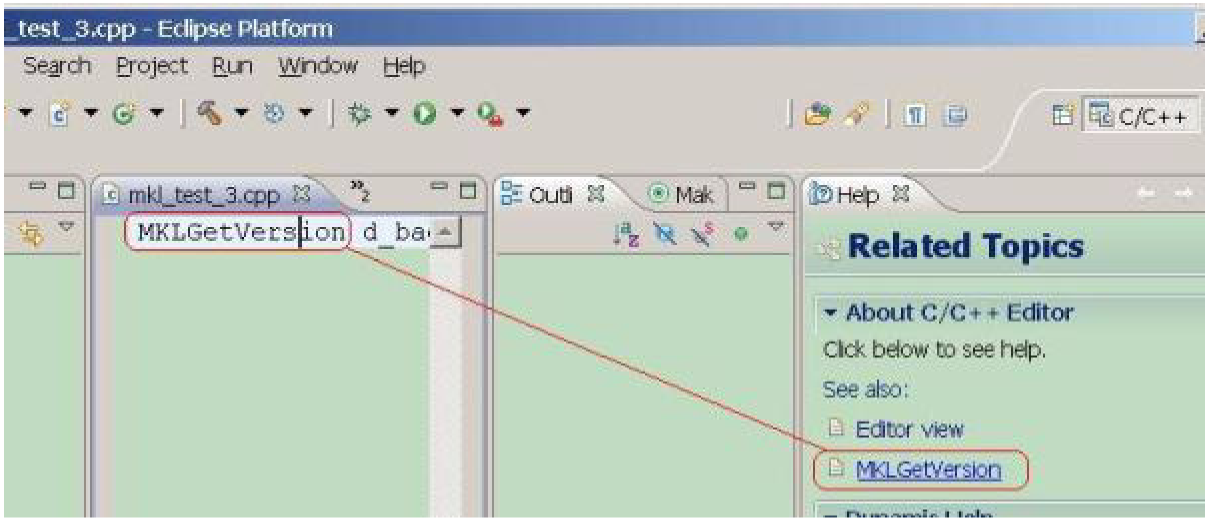
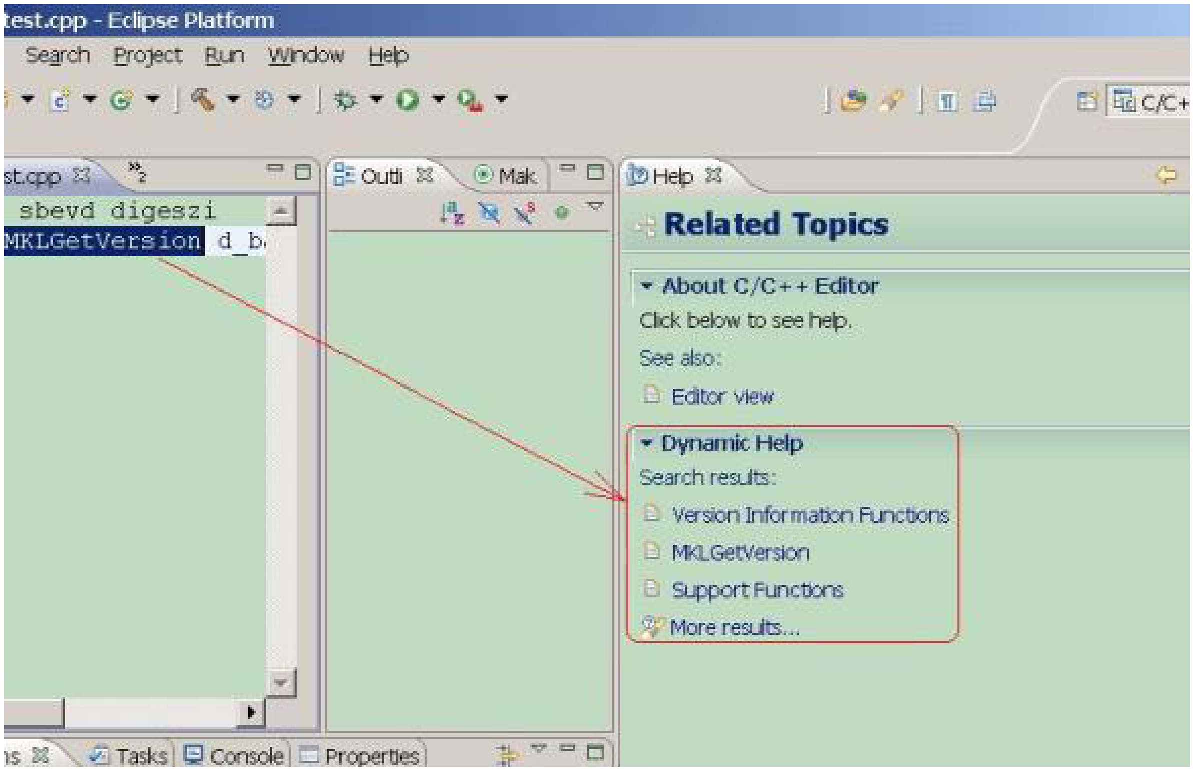


**Figure 10-5 F1 Help Search in the Eclipse* IDE CDT**



## Using Code/Content Assist in the Eclipse* IDE CDT

Code or Content Assist feature in Eclipse means automatic prompt for possible completion of a code line being written in the editor window.

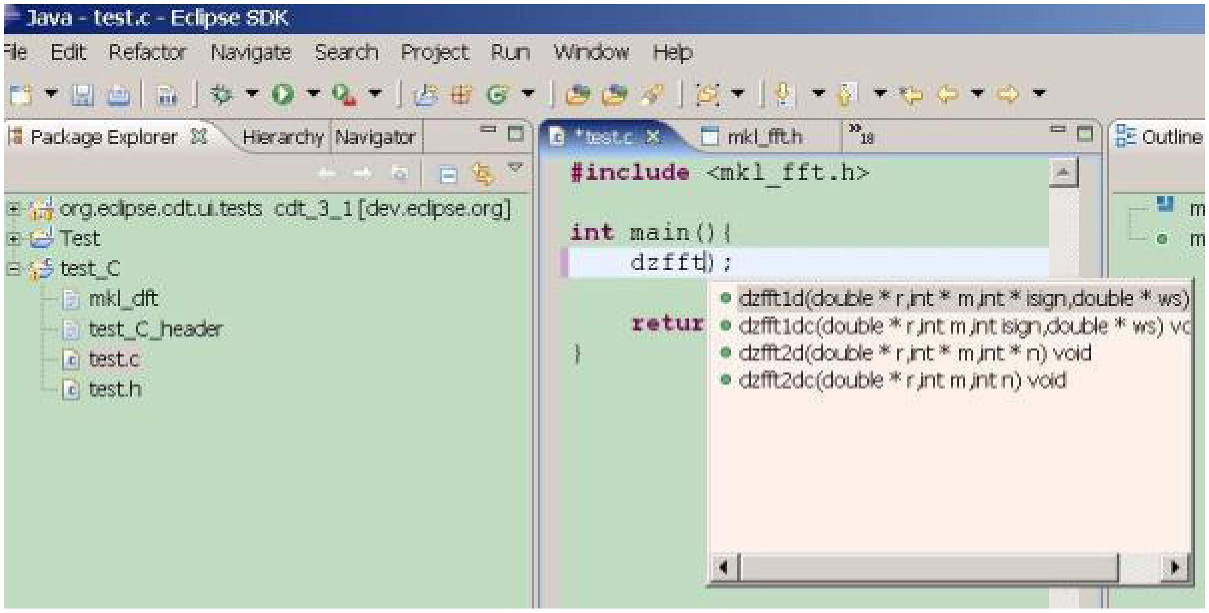
For a software library, Eclipse looks up possible completion of a function name or a named constant in a header file. So, to enable Code/Content Assist with Intel MKL,

1. Specify paths to include files as explained in Configuring the Eclipse* IDE CDT to Link with Intel MKL in chapter 4 or follow the recommendations given in the respective User's Guide on configuring the Eclipse CDT shell to link with the library.
2. Add the `#include` statement with the needed header file <header.h> to your source file.

To be prompted for the completion of the name of an Intel MKL function or a named constant in the code window,

1. Type the first few characters of the name in your code line
2. Press Ctrl + SPACEBAR.
   The prompt info appears in a popup.

**Figure 10-6 Code/Content Assist**


Java - test.c - Eclipse SDK
File Edit Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
Navigator
*test.c
mkl_fft.h
Outline
org.eclipse.cdt.ui.tests cdt_3_1 [dev.eclipse.org]
Test
test_C
mkl_dft
test_C_header
test.c
test.h

```
#include <mkl_fft.h>

int main(){
    dzfft);

    retur
}
```

```
dzfft1d(double * r,int * m,int * isign,double * ws)
dzfft1dc(double * r,int m,int isign,double * ws) vo
dzfft2d(double * r,int * m,int * n) void
dzfft2dc(double * r,int m,int n) void
```

## Customizing Code/Content Assist

To customize Code/Content Assist, set Content Assist preferences:

1. Click **Window > Preferences**
2. Expand **C/C++**, and click **C/C++ Editor**
3. Click the **Content Assist** tab
4. Do the following (see Figure 10-7):
   To change the scope from which Content Assist retrieves its proposals, select either **Search current file and included files** or **Search current project** by checking the appropriate the radio button

To insert an element if it is the only item in the list when Content Assist is invoked check **Insert single proposals automatically**

To display proposals in alphabetical order, rather than by relevance, check **Present proposals in alphabetical order**

To change the amount of time Content Assist is permitted to parse proposals type the value in the **Content Assist parsing timeout** text box area.

To enable alternative triggers for Content Assist, check appropriate check boxes under **Auto activation**

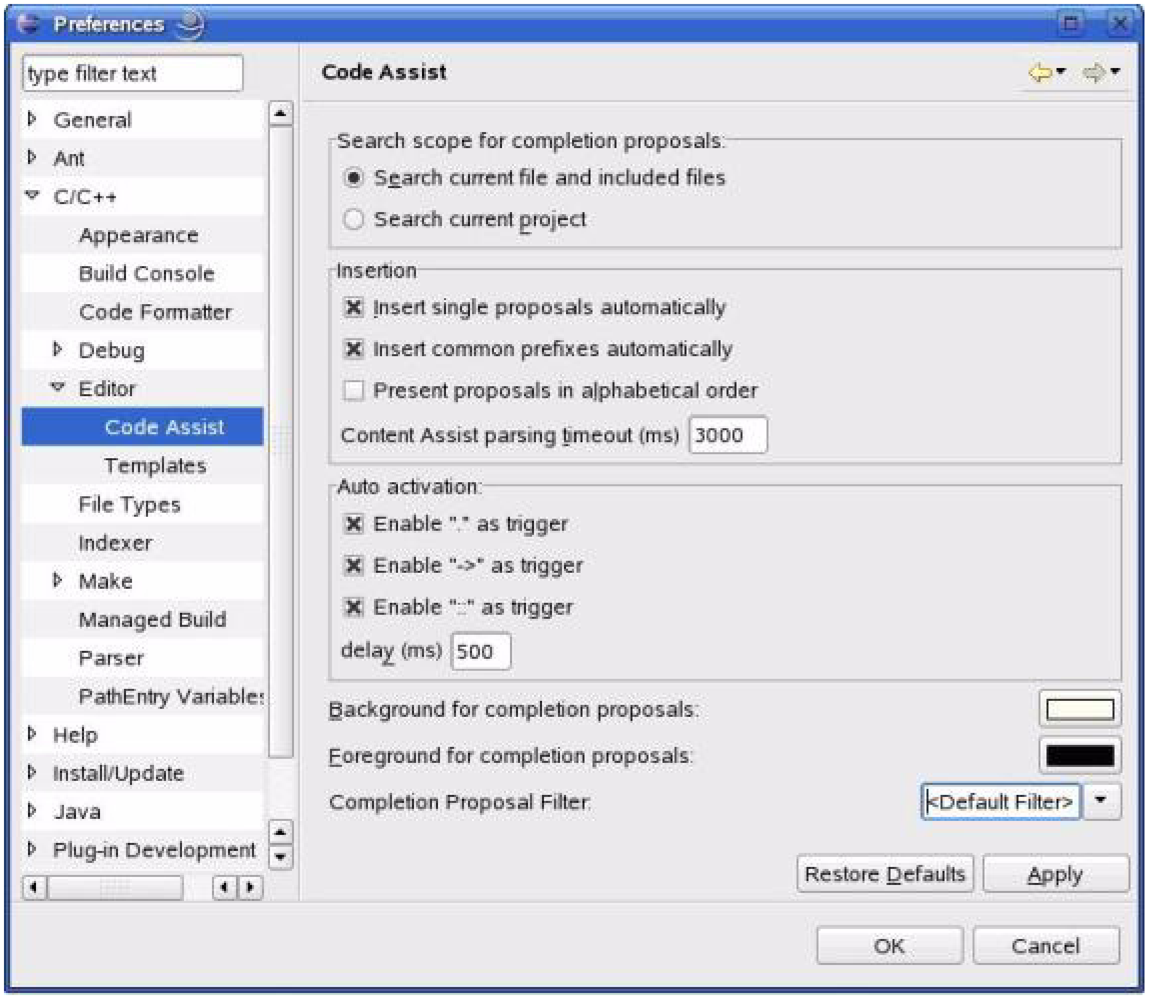
To change the delay before Content Assist is automatically invoked for the triggers (see above), type the new delay in the **delay** text box area under **Auto activation**.

To change the background color of the Content Assist dialog box, click the color palette button next to **Background for completion proposals**

To change the foreground color of the Content Assist dialog box, click the color palette button next to **Foreground for completion proposals**

5. Click **OK**.

**Figure 10-7 Customizing Code/Content Assist**



Code Assist

Search scope for completion proposals:

- [✓ Search current file and included files]
- Search current project

Insertion

- [✓ Insert single proposals automatically]
- [✓ Insert common prefixes automatically]
- Present proposals in alphabetical order

Content Assist parsing timeout (ms) 3000

Auto activation:

- [✓ Enable "." as trigger]
- [✓ Enable "->" as trigger]
- [✓ Enable "::" as trigger]

delay (ms) 500

Background for completion proposals: ⬜

Foreground for completion proposals: ⬛

Completion Proposal Filter: <Default Filter>

# LINPACK and MP LINPACK Benchmarks 11

This chapter describes the Intel® Optimized LINPACK Benchmark for the Linux* OS and Intel® Optimized MP LINPACK Benchmark for Clusters.

## Intel® Optimized LINPACK Benchmark for Linux OS*

Intel® Optimized LINPACK Benchmark is a generalization of the LINPACK 1000 benchmark. It solves a dense (`real`*8) system of linear equations (*Ax*=*b*), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate, and tests the results for accuracy. The generalization is in the number of equations (*N*) it can solve, which is not limited to 1000. It uses partial pivoting to assure the accuracy of the results.

This benchmark should not be used to report LINPACK 100 performance, as that is a compiled-code only benchmark. This is a shared memory (SMP) implementation which runs on a single platform and should not be confused with MP LINPACK, which is a distributed memory version of the same benchmark. This benchmark should not be confused with LINPACK, the library, which has been expanded upon by the LAPACK library.

Intel is providing optimized versions of the LINPACK benchmarks to make it easier than using HPL for you to obtain high LINPACK benchmark results on your systems based on genuine Intel® processors. Use this package to benchmark your SMP machine.

Additional information on this software as well as other Intel® software performance products is available at http://developer.intel.com/software/products/.

## Contents

The Intel Optimized LINPACK Benchmark for Linux* contains the following files, located in the `./benchmarks/linpack/` subdirectory in the Intel MKL directory (see Table 3-1):

**Table 11-1 Contents of the LINPACK Benchmark**

| **./benchmarks/linpack/** | |
|---|---|
| `linpack_itanium` | The 64-bit program executable for a system based on Intel® Itanium® 2 processor. |
| `linpack_xeon32` | The 32-bit program executable for a system based on Intel® Xeon® processor or Intel® Xeon® processor MP with or without Streaming SIMD Extensions 3 (SSE3). |
| `linpack_xeon64` | The 64-bit program executable for a system with Intel® Xeon® processor using Intel® 64 architecture. |
| `runme_itanium` | A sample shell script for executing a pre-determined problem set for `linpack_itanium`. `OMP_NUM_THREADS` set to 8 processors. |
| `runme_xeon32` | A sample shell script for executing a pre-determined problem set for `linpack_xeon32`. `OMP_NUM_THREADS` set to 2 processors. |
| `runme_xeon64` | A sample shell script for executing a pre-determined problem set for linpack_xeon64. `OMP_NUM_THREADS` set to 4 processors. |
| `lininput_itanium` | Input file for pre-determined problem for the `runme_itanium` script. |
| `lininput_xeon32` | Input file for pre-determined problem for the `runme_xeon32` script. |
| `lininput_xeon64` | Input file for pre-determined problem for the `runme_xeon64` script. |
| `lin_itanium.txt` | Result of the `runme_itanium` script execution. |
| `lin_xeon32.txt` | Result of the `runme_xeon32` script execution. |
| `lin_xeon64.txt` | Result of the `runme_xeon64` script execution. |
| `help.lpk` | Simple help file. |
| `xhelp.lpk` | Extended help file. |

## Running the Software

To obtain results for the pre-determined sample problem sizes on a given system, type one of the following, as appropriate:

```
./runme_itanium
./runme_xeon32
./runme_xeon64 .
```

To run the software for other problem sizes, please refer to the extended help included with the program. Extended help can be viewed by running the program executable with the "`-e`" option:

```
./xlinpack_itanium -e

./xlinpack_xeon32 -e

./xlinpack_xeon64 -e .
```

The pre-defined data input files `lininput_itanium`, `lininput_xeon32`, and `lininput_xeon64` are provided merely as examples. Different systems may have different number of processors, or amount of memory, and require new input files. The extended help can be used for insight into proper ways to change the sample input files.

Each input file requires at least the following amount of memory:

| | |
|---|---|
| `lininput_itanium` | 16 GB |
| `lininput_xeon32` | 2 GB |
| `lininput_xeon64` | 16 GB. |

If the system has less memory than the above sample data inputs require, you may have to edit or create your own data input files, as directed in the extended help.

Each sample script, in particular, uses the `OMP_NUM_THREADS` environment variable to set the number of processors it is targeting. To optimize performance on a different number of physical processors, change that line appropriately. If you run the Intel Optimized LINPACK Benchmark without setting the number of threads, it will default to the number of cores according to the OS. You can find the settings for this environment variable in the `runme_*` sample scripts. If the settings do not already match the situation for your machine, edit the script.

## Known Limitations

The following limitations are known for the Intel Optimized LINPACK Benchmark for Linux*:

- Intel Optimized LINPACK Benchmark is threaded to effectively use multiple processors. So, in multi-processor systems, best performance will be obtained with Hyper-Threading technology turned off, which ensures that the operating system assigns threads to physical processors only.
- If an incomplete data input file is given, the binaries may either hang or fault. See the sample data input files and/or the extended help for insight into creating a correct data input file.

# Intel® Optimized MP LINPACK Benchmark for Clusters

The Intel® Optimized MP LINPACK Benchmark for Clusters is based on modifications and additions to HPL 1.0a from Innovative Computing Laboratories (ICL) at the University of Tennessee, Knoxville (UTK). The benchmark can be used for Top 500 runs (see http://www.top500.org). The use of the benchmark requires that you are already intimately familiar with the HPL distribution and usage. This package adds some additional enhancements and bug fixes designed to make the HPL usage more convenient. The `./benchmarks/mp_linpack` directory adds techniques to minimize search times frequently associated with long runs.

The Intel® Optimized MP LINPACK Benchmark for Clusters is an implementation of the Massively Parallel MP LINPACK benchmark. HPL code was used as a basis. It solves a random dense (`real`*8) system of linear equations ($Ax=b$), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate and tests the results for accuracy. You can solve any size ($N$) system of equations that fit into memory. The benchmark uses full row pivoting to ensure the accuracy of the results.

This benchmark should not be used to report LINPACK performance on a shared memory machine. For that, the Intel® Optimized LINPACK Benchmark should be used instead. This benchmark should be used on a distributed memory machine.

Intel is providing optimized versions of the LINPACK benchmarks to make it easier than using HPL for you to obtain high LINPACK benchmark results on your systems based on genuine Intel® processors. Use this package to benchmark your cluster. The prebuilt binaries require Intel® MPI 3.x be installed on the cluster. The run-time version of Intel MPI is free and can be downloaded from www.intel.com/software/products/cluster.

**NOTE.** If you wish to use a different version of MPI, you can do so by using the MP LINPACK source provided.

The package includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories and neither the University nor ICL endorse or promote this product. Although HPL 1.0a is redistributable under certain conditions, this particular package is subject to the MKL license.

Intel MKL 10.0 Update 3 has introduced a new functionality into MP LINPACK, which is called a hybrid build, while continuing to support the older version. The term "*hybrid*" refers to special optimizations added to take advantage of mixed OpenMP*/MPI parallelism. If you want to use one MPI process per node and to achieve further parallelism via OpenMP, use of the hybrid build. If you want to rely exclusively on MPI for parallelism and use one MPI per core, use of the non-hybrid build. In addition to supplying certain hybrid prebuilt binaries, Intel MKL supplies certain hybrid prebuilt libraries to take advantage of the additional OpenMP optimizations.

Note that the non-hybrid version may be used in a hybrid mode, but it would be missing some of the optimizations added to the hybrid version. Non-hybrid builds are the default. In many cases, the use of the hybrid mode is required for system reasons, but if there is a choice, the non-hybrid code may be faster, although that may change in future releases. To use the non-hybrid code in a hybrid mode, use the threaded MPI and Intel MKL, link with a thread-safe MPI, and call function `MPI_init_thread()` so as to indicate a need for MPI to be thread-safe.

## Contents

The Intel Optimized MP LINPACK Benchmark for Clusters includes the HPL 1.0a distribution in its entirety as well as the modifications, delivered in the files listed in Table 11-2 and located in the `./benchmarks/mp_linpack/` subdirectory in the Intel MKL directory (see Table 3-1):

**Table 11-2 Contents of the MP LINPACK Benchmark**

| **./benchmarks/mp_linpack/** | |
|---|---|
| `testing/ptest/HPL_pdtest.c` | HPL 1.0a code modified to display captured `DGEMM` information in `ASYOUGO2_DISPLAY` (see details in the New Features section) if it was captured. |
| `src/blas/HPL_dgemm.c` | HPL 1.0a code modified to capture `DGEMM` information if desired from `ASYOUGO2_DISPLAY` |
| `src/grid/HPL_grid_init.c` | HPL 1.0a code modified to do additional grid experiments originally not in HPL 1.0. |
| `src/pgesv/HPL_pdgesvK2.c` | HPL 1.0a code modified to do `ASYOUGO` and `ENDEARLY` modifications |
| `include/hpl_misc.h and hpl_pgesv.h` | Bugfix added to allow for 64-bit address computation. |
| `src/pgesv/HPL_pdgesv0.c` | HPL 1.0a code modified to do `ASYOUGO`, `ASYOUGO2`, and `ENDEARLY` modifications |
| `testing/ptest/HPL.dat` | HPL 1.0a sample `HPL.dat` modified. |
| `Make.ia32` | (New) Sample architecture make for processors using IA-32 architecture and Linux. |
| `Make.em64t` | (New) Sample architecture make for processors using Intel® 64 architecture and Linux. |
| `Make.ipf` | (New) Sample architecture make for IA-64 architecture and Linux. |
| `HPL.dat` | A repeat of `testing/ptest/HPL.dat` in the top-level directory |

**Table 11-2 Contents of the MP LINPACK Benchmark**

| **./benchmarks/mp_linpack/** | |
|---|---|
| Next three files are prebuilt executables, readily available for simple performance testing. | |
| `bin_intel/ia32/xhpl_ia32` | (New) Prebuilt binary for IA-32 architecture, Linux, and Intel® MPI 3.0. |
| `bin_intel/em64t/xhpl_em64t` | (New) Prebuilt binary for Intel® 64 architecture, Linux, and Intel MPI 3.0. |
| `bin_intel/ipf/xhpl_ipf` | (New) Prebuilt binary for IA-64 architecture, Linux, and Intel MPI 3.0. |
| Next three files are prebuilt hybrid executables. | |
| `bin_intel/ia32/xhpl_hybrid_ia32` | (New) Prebuilt hybrid binary for IA-32 architecture, Linux, and Intel MPI 3.0. |
| `bin_intel/em64t/xhpl_hybrid_em64t` | (New) Prebuilt hybrid binary for Intel® 64 architecture, Linux, and Intel MPI 3.0. |
| `bin_intel/ipf/xhpl_hybrid_ipf` | (New) Prebuilt hybrid binary for IA-64 architecture, Linux, and Intel MPI 3.0. |
| | |
| `lib_hybrid/32/libhpl_hybrid.a` | (New) Prebuilt library with the hybrid version of MP LINPACK for IA-32 architecture. |
| `lib_hybrid/em64t/libhpl_hybrid.a` | (New) Prebuilt library with the hybrid version of MP LINPACK for Intel® 64 architecture. |
| `lib_hybrid/64/libhpl_hybrid.a` | (New) Prebuilt library with the hybrid version of MP LINPACK for IA-64 architecture. |
| `nodeperf.c` | (New) Sample utility that tests the `DGEMM` speed across the cluster. |

## Building MP LINPACK

There are a few included sample architecture makes. It is recommended that you edit them to fit your specific configuration. In particular:

- Set `TOPdir` to the directory MP LINPACK is being built in.
- You may set MPI variables, that is, `MPdir`, `MPinc`, and `MPlib`.
- Specify the location of Intel MKL and of files to be used (`LAdir`, `LAinc`, `LAlib`).
- Adjust compiler and compiler/linker options.
- Specify the version of MP LINPACK you are going to build (hybrid or non-hybrid) by setting the version parameter for the make, for example,

```
make arch=em64t version=hybrid install
```

For some sample cases, like Linux systems based on Intel® 64 architecture, the makes contain values that seem to be common. However, you are required to be familiar with building HPL and picking appropriate values for these variables.

## New Features

The toolset is basically identical with the HPL 1.0a distribution. There are a few changes which are optionally compiled in and are disabled until you specifically request them. These new features are:

**ASYOUGO:** Provides non-intrusive performance information while runs proceed. There are only a few outputs and this information does not impact performance. This is especially useful because many runs can go hours without any information.

**ASYOUGO2:** Provides slightly intrusive additional performance information because it intercepts every `DGEMM`.

**ASYOUGO2_DISPLAY:** Displays the performance of all the significant `DGEMMs` inside the run.

**ENDEARLY:** Displays a few performance hints and then terminates the run early.

**FASTSWAP:** Inserts the LAPACK-optimized `DLASWP` into HPL's code. This may yield a benefit for Itanium® 2 processor. You can experiment with this to determine best results.

**HYBRID:** Establishes the Hybrid OpenMP/MPI mode of MP LINPACK, providing the possibility to use threaded Intel MKL and prebuilt MP LINPACK hybrid libraries.

**WARNING.** Use this option only with an Intel compiler and the Intel® MPI library version 3.1 or higher. You are also recommended to use the compiler version 10.0 or higher.

## Benchmarking a Cluster

To benchmark a cluster, follow the sequence of steps (maybe, optional) below. Pay special attention to the iterative steps 3 and 4. They make up a loop that searches for HPL parameters (specified in `HPL.dat`) which the top performance of you cluster is reached with.

1. Get HPL installed and functional on all the nodes.
2. You may run `nodeperf.c` (included in the distribution) to see the performance of `DGEMM` on all the nodes.

   Compile `nodeperf.c` in with your MPI and Intel MKL.

   For example,

   ```
   mpicc -O3 nodeperf.c <mkl_directory>/lib/em64t/libmkl_em64t.a
   <mkl_directory>/lib/em64t/libguide.a -lpthread -o nodeperf .
   ```
   Launching `nodeperf.c` on all the nodes is especially helpful in a very large cluster. Indeed, there may be a stray job on a certain node, for example, 738, which is running 5% slower than the rest. MP LINPACK will then run as slow as the slowest node. In this case, `nodeperf` enables quick identifying of the potential problem spot without lots of small MP LINPACK runs around the cluster in search of the bad node. It is common that after a bunch of HPL runs, there may be zombie processes and `nodeperf` facilitates finding the slow nodes. It goes through all the nodes, one at a time, and reports the performance of `DGEMM` followed by some host identifier. Therefore, the higher the penultimate number then, the faster that node was performing.
3. Edit `HPL.dat` to fit your cluster needs.

   Read through the HPL documentation for ideas on this. However, you should try on at least 4 nodes.
4. Make an HPL run, using compile options such as `ASYOUGO` or `ASYOUGO2` or `ENDEARLY` to aid in your search (These options enable you to gain insight into the performance sooner than HPL would normally give this insight.)

   When doing so, follow these recommendations:

   — Use the MP LINPACK patched version of HPL to save time in the searching.

     Using a patched version of HPL should not hinder your performance. That's why features that could be performance intrusive are compile-optional (and it is called out below) in MP LINPACK. That is, if you don't use any of the new options explained in section Options to reduce search time, then these changes are disabled. The primary purpose of the additions is to assist you in finding solutions.

     HPL requires long time to search for many different parameters. In the MP LINPACK, the goal is to get the best possible number.

     Given that the input is not fixed, there is a large parameter space you must search over. In fact, an exhaustive search of all possible inputs is improbably large even for a powerful cluster.

     This patched version of HPL optionally prints information on performance as it proceeds, or even terminates early depending on your desires.

   — Save time by compiling with `-DENDEARLY -DASYOUGO2` (described in the Options to reduce search time section) and using a negative threshold (Do not to use a negative threshold on the final run that you intend to submit if you are doing a Top500 entry!) You can set the threshold in line 13 of the HPL 1.0a input file `HPL.dat`.

   — If you are going to run a problem to completion, do it with `-DASYOUGO` (see Options to reduce search time section).

5. Using the quick performance feedback, return to step 3 and iterate until you are sure that the performance is as good as possible.

## Options to reduce search time

Running huge problems to completion on large numbers of nodes can take many hours. The search space for MP LINPACK is also huge: not only can you run any size problem, but over a number of block sizes, grid layouts, lookahead steps, using different factorization methods, etc. It can be a large waste of time to run a huge problem to completion only to discover it ran 0.01% slower than your previous best problem.

There are 3 options you might want to experiment with to reduce the search time:

- `-DASYOUGO`
- `-DENDEARLY`
- `-DASYOUGO2`

  Use cautiously, as it does have a marginal performance impact. To see `DGEMM` internal performance, compile with `-DASYOUGO2` and `-DASYOUGO2_DISPLAY`. This will give lots of useful `DGEMM` performance information at the cost of around 0.2% performance loss.

If you want the old HPL back, simply don't define these options and recompile from scratch (try "`make arch=<arch> clean_arch_all`").

**-DASYOUGO:** Gives performance data as the run proceeds. The performance always starts off higher and then drops because this actually happens in LU decomposition. The `ASYOUGO` performance estimate is usually an overestimate (because LU slows down as it goes), but it gets more accurate as the problem proceeds. The greater the lookahead step, the less accurate the first number may be. `ASYOUGO` tries to estimate where one is in the LU decomposition that MP LINPACK performs and this is always an overestimate as compared to `ASYOUGO2`, which measures actually achieved `DGEMM` performance. Note that the `ASYOUGO` output is a subset of the information that `ASYOUGO2` provides. So, refer to the description of the `-DASYOUGO2` option below for the details of the output.

**-DENDEARLY:** Terminates the problem after a few steps, so that you can set up 10 or 20 HPL runs without monitoring them, see how they all do, and then only run the fastest ones to completion. `-DENDEARLY` assumes `-DASYOUGO`. You do not need to define both, although it doesn't hurt. Because the problem terminates early, it is recommended setting the "threshold" parameter in `HPL.dat` to a negative number when testing `ENDEARLY`. There is no point in doing a residual check if the problem ended early. It also sometimes gives a better picture to compile with `-DASYOUGO2` when using `-DENDEARLY`.

You need to know the specifics of `-DENDEARLY`:

- `-DENDEARLY` stops the problem after a few iterations of `DGEMM` on the blocksize (the bigger the blocksize, the further it gets). It prints only 5 or 6 "updates", whereas `-DASYOUGO` prints about 46 or so outputs before the problem completes.
- Performance for `-DASYOUGO` and `-DENDEARLY` always starts off at one speed, slowly increases, and then slows down toward the end (because that is what LU does). `-DENDEARLY` is likely to terminate before it starts to slow down.
- `-DENDEARLY` terminates the problem early with an HPL Error exit. It means that you need to ignore the missing residual results, which are wrong, as the problem never completed. However, you can get an idea what the initial performance was, and if it looks good, then run the problem to completion without `-DENDEARLY`. To avoid the error check, you can set HPL's threshold parameter in `HPL.dat` to a negative number.
- Though `-DENDEARLY` terminates early, HPL treats the problem as completed and computes Gflop rating as though the problem ran to completion. Ignore this erroneously high rating.
- The bigger the problem, the more accurately the last update that `-DENDEARLY` returns will be close to what happens when the problem runs to completion. `-DENDEARLY` is a poor approximation for small problems. It is for this reason that you are suggested to use `ENDEARLY` in conjunction with `ASYOUGO2`, because `ASYOUGO2` reports actual `DGEMM` performance, which can be a closer approximation to problems just starting.

The best known compile options for Itanium® 2 processor are with the Intel® compiler and look like this:

```
-O2 -ipo -ipo_obj -ftz -IPF_fltacc -IPF_fma -unroll -w -tpp2
```

**-DASYOUGO2:** Gives detailed single-node `DGEMM` performance information. It captures all `DGEMM` calls (if you use Fortran BLAS) and records their data. Because of this, the routine has a marginal intrusive overhead. Unlike `-DASYOUGO`, which is quite non-intrusive, `-DASYOUGO2` is interrupting every `DGEMM` call to monitor its performance. You should beware of this overhead, although for big problems, it is, for sure, less than 1/10th of a percent.

Here is a sample `ASYOUGO2` output (the first 3 non-intrusive numbers can be found in `ASYOUGO` and `ENDEARLY`), so it suffices to describe these numbers here:

```
Col=001280 Fract=0.050 Mflops=42454.99 (DT= 9.5 DF= 34.1
DMF=38322.78).
```

The problem size was *N*=16000 with a blocksize of 128. After 10 blocks, that is, 1280 columns, an output was sent to the screen. Here, the fraction of columns completed is 1280/16000=0.08. Only about 20 outputs are printed, at various places through the matrix decomposition: `fractions 0.005,0.010,0.015,0.02,0.025,0.03,0.035, 0.04,0.045,0.05,0.055,0.06,0.065,0.07,0.075,0.080,0.085,0.09,0.095,. 10,...,.195,.295,.395,...,.895`. However, this problem size is so small and the block size so big by comparison that as soon as it printed the value for `0.045`, it was already through 0.08 fraction of the columns. On a really big problem, the fractional number will be more accurate. It never prints more than the 46 numbers above. So, smaller problems will have fewer than 46 updates, and the biggest problems will have precisely 46 updates.

The `Mflops` is an estimate based on 1280 columns of LU being completed. However, with lookahead steps, sometimes that work is not actually completed when the output is made. Nevertheless, this is a good estimate for comparing identical runs.

The 3 numbers in parenthesis are intrusive `ASYOUGO2` addins. The `DT` is the total time processor 0 has spent in `DGEMM`. The `DF` is the number of billion operations that have been performed in `DGEMM` by one processor. Hence, the performance of processor 0 (in Gflops) in `DGEMM` is always `DF/DT`. Using the number of `DGEMM` flops as a basis instead of the number of LU flops, you get a lower bound on performance of our run by looking at `DMF`, which can be compared to `Mflops` above (It uses the global LU time, but the `DGEMM` flops are computed under the assumption that the problem is evenly distributed amongst the nodes, as only HPL's node (0,0) returns any output.)

Note that when using the above performance monitoring tools to compare different `HPL.dat` inputs, you should beware that the pattern of performance drop off that LU experiences is sensitive to some of the inputs. For instance, when you try very small problems, the performance drop off from the initial values to end values is very rapid. The larger the problem, the less the drop off, and it is probably safe to use the first few performance values to estimate the difference between a problem size 700000 and 701000, for instance. Another factor that influences the performance drop off is the grid dimensions (P and Q). For big problems, the performance tends to fall off less from the first few steps when P and Q are roughly equal in value. You can make use of a large number of parameters, such as broadcast types, and change them so that the final performance is determined very closely by the first few steps.

Using these tools will greatly assist the amount of data you can test.

# *Intel® Math Kernel Library Language Interfaces Support* A

Table A-1 shows language interfaces that Intel® Math Kernel Library (Intel® MKL) provides for each function domain, and Table A-2 lists the respective header files. However, Intel MKL routines can be called from other languages using mixed-language programming. For example, see section "Mixed-language programming with Intel® MKL" in chapter 7 on how to call Fortran routines from C/C++.

**Table A-1 Intel® MKL language interfaces support**

| Function Domain | FORTRAN 77 interface | Fortran 90/95 interface | C/C++ interface |
|---|---|---|---|
| Basic Linear Algebra Subprograms (BLAS) | + | + | via CBLAS |
| BLAS-like extension transposition routines | + |  | + |
| Sparse BLAS Level 1 | + | + | via CBLAS |
| Sparse BLAS Level 2 and 3 | + | + | + |
| LAPACK routines for solving systems of linear equations | + | + | * |
| LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations | + | + | * |
| Auxiliary and utility LAPACK routines | + |  | * |
| Parallel Basic Linear Algebra Subprograms (PBLAS) | + |  |  |
| ScaLAPACK routines | + |  | * |
| DSS/PARDISO* solvers | + | + | + |
| Other Direct and Iterative Sparse Solver routines | + | + | + |
| Vector Mathematical Library (VML) functions | + | + | + |
| Vector Statistical Library (VSL) functions | + | + | + |
| Fourier Transform functions (FFT) |  | + | + |
| Cluster FFT functions |  | + | + |

**Table A-1 Intel® MKL language interfaces support** (continued)

| Function Domain | FORTRAN 77 interface | Fortran 90/95 interface | C/C++ interface |
|---|---|---|---|
| Trigonometric Transform routines | | + | + |
| Fast Poisson, Laplace, and Helmholtz Solver (Poisson Library) routines | | + | + |
| Optimization (Trust-Region) Solver routines | + | + | + |
| GMP* arithmetic functions | | | + |
| Service routines (including memory allocation) | | | + |

* Supported using a mixed language programming call. See Table A-2 for the respective header file.

Table A-2 lists available header files for all Intel MKL function domains.

**Table A-2 Intel® MKL include files**

| Function domain | Include files | |
|---|---|---|
| | Fortran | C or C++ |
| **All function domains** | `mkl.fi` | `mkl.h` |
| BLAS Routines | `mkl_blas.f90`<br>`mkl_blas.fi` | `mkl_blas.h` |
| BLAS-like Extension Transposition Routines | `mkl_trans.fi` | `mkl_trans.h` |
| CBLAS Interface to BLAS | | `mkl_cblas.h` |
| Sparse BLAS Routines | `mkl_spblas.fi` | `mkl_spblas.h` |
| LAPACK Routines | `mkl_lapack.f90`<br>`mkl_lapack.fi` | `mkl_lapack.h` |
| ScaLAPACK Routines | | `mkl_scalapack.h` |
| All Sparse Solver Routines | `mkl_solver.f90` | `mkl_solver.h` |
| • PARDISO | `mkl_pardiso.f77`<br>`mkl_pardiso.f90` | `mkl_pardiso.h` |
| • DSS Interface | `mkl_dss.f77`<br>`mkl_dss.f90` | `mkl_dss.h` |
| • RCI Iterative Solvers<br>• ILU Factorization | `mkl_rci.fi` | `mkl_rci.h` |
| Optimization Solver Routines | `mkl_rci.fi` | `mkl_rci.h` |
| Vector Mathematical Functions | `mkl_vml.f77`<br>`mkl_vml.fi` | `mkl_vml.h` |

**Table A-2 Intel® MKL include files** (continued)

| Function domain | Include files | |
|---|---|---|
| | **Fortran** | **C or C++** |
| Vector Statistical Functions | `mkl_vml.f77`<br>`mkl_vsl.fi` | `mkl_vsl.h` |
| Fourier Transform Functions | `mkl_dfti.f90` | `mkl_dfti.h` |
| Cluster Fourier Transform Functions | `mkl_cdft.f90` | `mkl_cdft.h` |
| Partial Differential Equations Support Routines | | |
| • Trigonometric Transforms | `mkl_trig_transforms.f90` | `mkl_trig_transforms.h` |
| • Poisson Solvers | `mkl_poisson.f90` | `mkl_poisson.h` |
| GMP interface | | `mkl_gmp.h` |
| Service routines | | `mkl_service.h` |
| Memory allocation routines | | `i_malloc.h` |
| MKL examples interface | | `mkl_example.h` |

# Support for Third-Party Interfaces

B

This appendix describes in brief certain interfaces that Intel® Math Kernel Library (Intel® MKL) supports.

## GMP* Functions

Intel MKL implementation of GMP* arithmetic functions includes arbitrary precision arithmetic operations on integer numbers. The interfaces of such functions fully match the GNU Multiple Precision* (GMP) Arithmetic Library. For specifications of these functions, please see http://www.intel.com/software/products/mkl/docs/gnump/WebHelp/.

If you currently use the GMP* library, you need to modify `INCLUDE` statements in your programs to `mkl_gmp.h`.

## FFTW Interface Support

Intel MKL offers two collections of wrappers being the FFTW interface (www.fftw.org) superstructure to be used for calling the Intel MKL Fourier transform functions. These collections correspond to the FFTW versions 2.x and 3.x and the Intel MKL versions 7.0 and later.

The purpose of these wrappers is to enable developers whose programs currently use FFTW to gain performance with the Intel MKL Fourier transforms without changing the program source code. See the *"FFTW to Intel® MKL Wrappers" appendix in the Intel MKL Reference Manual* (file `mklman.pdf`) for details on the use of the wrappers.

# *Index*